

**UNISYS**

OS 1100

Meta-Assembler  
(MASM)

**Programming  
Reference Manual**

Copyright © 1992 Unisys Corporation.  
All rights reserved.  
Unisys is a registered trademark of Unisys Corporation.

Release SB4R4

July 1992

Priced Item

Printed in U S America  
7830 8269-001

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the Business Reply Mail form in this document, or remarks may be addressed directly to Unisys Corporation, PI Response Card, P.O. Box 64942, St. Paul, Minnesota, 55164-9749, U.S.A.

RESTRICTED—Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.227-7013 and 252.211-7015/FAR 52.227-14 and 52.227-19 for commercial computer software, as applicable.

# Contents

<b>About This Manual</b> .....	xv
 <b>Section 1. Introduction</b>	
<b>1.1. Two-Pass Assembler (Summary and Generative)</b> ..	1-2
<b>1.2. Dictionary</b> .....	1-2
<b>1.3. New Features and Enhancements</b> .....	1-3
<b>1.4. Summary of MASM Directives and Functions</b> .....	1-4
 <b>Section 2. MASM Usage</b>	
<b>2.1. Processor Call</b> .....	2-1
2.1.1. Processor Options .....	2-1
2.1.2. Reusability .....	2-5
<b>2.2. Output</b> .....	2-6
2.2.1. Output Fields .....	2-6
2.2.2. Preamble Output .....	2-7
2.2.3. END MASM Line .....	2-7
2.2.4. Cross-Reference Listing .....	2-9
<b>2.3. Input</b> .....	2-14
2.3.1. Statements .....	2-14
2.3.2. Symbols .....	2-15
2.3.3. Fields .....	2-16
2.3.4. Line Continuation .....	2-23
<b>2.4. Library Searching</b> .....	2-24
2.4.1. Procedure Library Search Chain .....	2-25
2.4.2. Search Order .....	2-25
<b>2.5. Procedure Library Search Table</b> .....	2-27
 <b>Section 3. Listing Control</b>	
<b>3.1. \$UNLIST — Inhibit Listing</b> .....	3-1
<b>3.2. \$LIST — Resume Listing</b> .....	3-1
<b>3.3. \$DISPLAY — Display Information</b> .....	3-1
<b>3.4. Page Ejection</b> .....	3-3

<b>3.5.</b>	<b>\$OCTAL — Set Binary Representation to Octal . . . .</b>	<b>3-3</b>
<b>3.6.</b>	<b>\$HEX — Set Binary Representation to Hexadecimal</b>	<b>3-3</b>
<b>3.7.</b>	<b>\$CROSSREF — Tailored Cross-Reference Listing . .</b>	<b>3-4</b>
<b>3.8.</b>	<b>\$HDG — Specify Print Heading . . . . .</b>	<b>3-5</b>
<b>3.9.</b>	<b>Listing Control Example . . . . .</b>	<b>3-7</b>

## Section 4. Values and Expressions

<b>4.1.</b>	<b>\$EQU — Equate a Value . . . . .</b>	<b>4-1</b>
<b>4.2.</b>	<b>Values . . . . .</b>	<b>4-2</b>
4.2.1.	Integer Values . . . . .	4-2
4.2.2.	Floating-Point Values . . . . .	4-8
4.2.3.	String Values . . . . .	4-9
4.2.4.	Nodes and Selectors . . . . .	4-18
4.2.5.	Control Information . . . . .	4-28
<b>4.3.</b>	<b>Expressions and Operators . . . . .</b>	<b>4-29</b>
4.3.1.	Operators . . . . .	4-29
4.3.2.	Expressions . . . . .	4-37
<b>4.4.</b>	<b>Data Types . . . . .</b>	<b>4-40</b>
4.4.1.	\$TYPE(e) — Compute Data Type Number . . . . .	4-40
4.4.2.	Type Testing Functions . . . . .	4-40
4.4.3.	\$IBITS(e) — Indicator Bits for Expression . . . . .	4-41

## Section 5. Data Generation

<b>5.1.</b>	<b>Signed Character Strings . . . . .</b>	<b>5-2</b>
<b>5.2.</b>	<b>Unsigned Character Strings . . . . .</b>	<b>5-3</b>
<b>5.3.</b>	<b>\$GEN — Data Generation . . . . .</b>	<b>5-4</b>
<b>5.4.</b>	<b>Forms . . . . .</b>	<b>5-5</b>
5.4.1.	\$FORM — Define a Form . . . . .	5-5
5.4.2.	\$GFORM — Generalized Form . . . . .	5-6
<b>5.5.</b>	<b>Location Counter Specification . . . . .</b>	<b>5-7</b>
5.5.1.	\$(e) — Location Counter Value . . . . .	5-7
5.5.2.	\$RES — Reserve Space . . . . .	5-7
5.5.3.	\$LCN — Location Counter Number . . . . .	5-7
5.5.4.	\$ILCN — Initial Location Counter Number . . . . .	5-8
5.5.5.	\$LCV(e) — Location Counter Value . . . . .	5-8
5.5.6.	\$LCB(e) — Location Counter Base . . . . .	5-8
5.5.7.	\$LCFV(e) — Location Counter Final Value . . . . .	5-8
<b>5.6.</b>	<b>\$LIT — Literal Pool Definition . . . . .</b>	<b>5-10</b>
<b>5.7.</b>	<b>\$WRD — Specify Word Size . . . . .</b>	<b>5-10</b>
<b>5.8.</b>	<b>\$NEG — Transform Negative Values . . . . .</b>	<b>5-11</b>

## Section 6. Assembly-Time Controls

<b>6.1. Condition Testing Directives</b>	6-1
6.1.1. \$IF — Conditional Interpretation	6-1
6.1.2. \$ELSE — Conditional Interpretation Alternative	6-2
6.1.3. \$ENDF — End Conditional Interpretation Group	6-2
6.1.4. \$ELSF — Conditional Interpretation Conditional Alternative	6-2
6.1.5. \$ANDF — And If	6-4
<b>6.2. Looping Directives</b>	6-6
6.2.1. \$DO — Repetitive Generation of a Line	6-6
6.2.2. \$ENDD — End \$DO Iteration	6-6
6.2.3. \$REPEAT — Repeat a Statement Group	6-7
6.2.4. \$ENDR — End \$REPEAT Construction	6-7
6.2.5. \$ENDI — End \$REPEAT Iteration	6-8
<b>6.3. \$NIL — No Action</b>	6-9
<b>6.4. \$GO — Transfer to a Name</b>	6-9
<b>6.5. \$NAME — Define an Internal Name</b>	6-10
<b>6.6. \$INSERT — Insert Images</b>	6-10
<b>6.7. Procedures and Functions</b>	6-11
6.7.1. Procedures	6-11
6.7.2. Functions	6-14
6.7.3. Nesting of Procedures	6-16
6.7.4. Waiting Labels	6-17
6.7.5. \$LF(e) — Label Field Description	6-18
6.7.6. Location Counter Control in Procedures	6-19
6.7.7. Use of \$NAME and \$GO Directives	6-20
6.7.8. Types of Procedures	6-21
6.7.9. Speeding Up a Two-Pass Procedure	6-28
6.7.10. \$FN(e1,e2) — Form a Name	6-30
6.7.11. Pass Initialization	6-30
<b>6.8. Microstrings</b>	6-31
<b>6.9. Levelers</b>	6-33

## Section 7. Dictionary Control

<b>7.1. Structure of the Dictionary</b>	7-2
<b>7.2. \$DELETE — Delete a Definition</b>	7-3
<b>7.3. \$HASH(e) — Dictionary Class Index</b>	7-4
<b>7.4. \$IC(e) — Identifier Class</b>	7-5
<b>7.5. \$LEVEL — Dictionary Level Control</b>	7-6
<b>7.6. \$LEV — Principal Dictionary Level</b>	7-7
<b>7.7. \$XLEV — Current Dictionary Level</b>	7-8

<b>7.8. Example — Value Retrieval .....</b>	<b>7-10</b>
---	-------------

## **Section 8. Processor Information Functions**

<b>8.1. \$DATE — Date and Time Function .....</b>	<b>8-1</b>
<b>8.2. \$LINES — Line Counter .....</b>	<b>8-2</b>
<b>8.3. \$PAR(e) — Processor Call Parameter .....</b>	<b>8-2</b>
<b>8.4. \$TMODES — MASM Operating Mode .....</b>	<b>8-3</b>

## **Section 9. MASM Output**

<b>9.1. Types of MASM Output .....</b>	<b>9-1</b>
<b>9.2. \$REL — Relocatable Binary Output .....</b>	<b>9-1</b>
<b>9.3. \$INFO — Special Information .....</b>	<b>9-2</b>
9.3.1. Processor Mode Settings (Group 1) .....	9-2
9.3.2. Common Block (Group 2) .....	9-3
9.3.3. Minimum D-Bank Specification (Group 3) .....	9-3
9.3.4. Blank Common Block (Group 4) .....	9-4
9.3.5. External Reference Definition (Group 5) .....	9-4
9.3.6. Entry-Point Definition (Group 6) .....	9-4
9.3.7. Even Starting Address (Group 7) .....	9-5
9.3.8. Static Diagnostic Information (Group 8) .....	9-5
9.3.9. Read-Only Location Counters (Group 9) .....	9-5
9.3.10. Extended Mode Location Counter (Group 10) .....	9-6
9.3.11. Void Bank (Group 11) .....	9-6
9.3.12. Library Search File (Group 12) .....	9-7
9.3.13. Restrictions .....	9-7

## **Section 10. Object Modules**

<b>10.1. \$OBJ — Select Object Module Output .....</b>	<b>10-2</b>
<b>10.2. Bank Groups .....</b>	<b>10-2</b>
<b>10.3. Banks (Location Counters) .....</b>	<b>10-3</b>
10.3.1. Bank Attributes .....	10-3
10.3.2. \$BANK Directive — Define Attributes for a Bank .....	10-13
10.3.3. Bank Name .....	10-15
10.3.4. Default Values for Banks .....	10-15
10.3.5. \$BANK Example .....	10-16
<b>10.4. References .....</b>	<b>10-17</b>
10.4.1. Reference Attributes .....	10-17
10.4.2. \$IMPORT Directive — Define Attributes for a Reference .....	10-21
10.4.3. Library Search Chains .....	10-22
10.4.4. Default Values for References .....	10-22

10.4.5.	\$IMPORT Directive Example .....	10-23
<b>10.5.</b>	<b>Definitions .....</b>	<b>10-24</b>
10.5.1.	Definition Attributes .....	10-24
10.5.2.	\$EXPORT Directive — Define Attributes for a Definition .....	10-27
10.5.3.	Default Values for Definitions .....	10-28
10.5.4.	\$EXPORT Directive Example .....	10-28
<b>10.6.</b>	<b>Link Vectors .....</b>	<b>10-29</b>
10.6.1.	Basic Mode .....	10-29
10.6.2.	Extended Mode .....	10-30
10.6.3.	Extended Mode Example .....	10-31
10.6.4.	Standard Procedures for the Creation of Link Vectors .....	10-32
<b>10.7.</b>	<b>Defaults .....</b>	<b>10-35</b>
<b>10.8.</b>	<b>Library Definition Element .....</b>	<b>10-36</b>
<b>10.9.</b>	<b>Compatibility .....</b>	<b>10-39</b>
10.9.1.	\$INFO Directives .....	10-39
10.9.2.	Collector-Defined Symbols .....	10-40
10.9.3.	Bank Switching .....	10-40
10.9.4.	Entry-Point START\$ .....	10-40

## Section 11. Symbolic Output

<b>11.1.</b>	<b>\$SYM — Establish Symbolic Output Mode .....</b>	<b>11-1</b>
<b>11.2.</b>	<b>\$OUTPUT — Output Symbolic Images .....</b>	<b>11-2</b>

## Section 12. Saving Definitions

<b>12.1.</b>	<b>Definition Mode Assembly .....</b>	<b>12-1</b>
<b>12.2.</b>	<b>\$DEF — Establish Definition Mode .....</b>	<b>12-3</b>
<b>12.3.</b>	<b>\$INCLUDE — Include Definitions .....</b>	<b>12-3</b>

## Section 13. Error and Warning Diagnostics

## Appendix A. OS 1100 Features

<b>A.1.</b>	<b>Instruction Mnemonic Redefinition .....</b>	<b>A-1</b>
<b>A.2.</b>	<b>Extended and Basic Mode Operation .....</b>	<b>A-2</b>
A.2.1.	Mode Directives .....	A-2
A.2.2.	No Mode Operation .....	A-3
A.2.3.	Basic Mode Operation .....	A-4
A.2.4.	Extended Mode Operation .....	A-5
A.2.5.	\$EXTEND Mode Block Transfer (BT) .....	A-8

A.2.6.	Instruction Generation .....	A-9
A.2.7.	\$BASE Directive .....	A-10
A.2.8.	\$USE Directive .....	A-13
A.2.9.	Literals .....	A-14
A.2.10.	\$BREG Function .....	A-15
A.2.11.	I\$ and EI\$ Forms .....	A-17
A.2.12.	\$EQU Directive (Equate a Field) .....	A-17
A.2.13.	\$PROC Directive .....	A-20
A.2.14.	\$FORCE and \$FORCEOFF Directives .....	A-21
A.2.15.	\$MACH Directive .....	A-22
A.2.16.	\$TMACH Function .....	A-24

## Appendix B. Object Module Evolution

<b>B.1.</b>	<b>OM\$DEF Name Changes (Formerly OMDEF\$) .....</b>	<b>B-2</b>
B.1.1.	Attribute Name Changes .....	B-2
B.1.2.	New Attribute Names .....	B-4
B.1.3.	\$FORM Name Changes .....	B-5
B.1.4.	Standard Definition Name Changes .....	B-5
B.1.5.	Procedure Name Changes .....	B-6
<b>B.2.</b>	<b>MASM 4R2 Object Module Changes .....</b>	<b>B-7</b>
B.2.1.	\$BANK (Bank Field Changes) .....	B-7
B.2.2.	\$IMPORT (Reference Field Changes and Extension) .....	B-8
B.2.3.	\$EXPORT (Definition Field Changes) .....	B-9
<b>B.3.</b>	<b>MASM 5R1 Object Module Changes .....</b>	<b>B-10</b>
B.3.1.	\$BANK (Bank Field Changes) .....	B-10
B.3.2.	\$IMPORT (Reference Field Changes) .....	B-11
B.3.3.	\$EXPORT (Definition Field Changes) .....	B-11
B.3.4.	Miscellaneous Changes .....	B-11
<b>B.4.</b>	<b>MASM 6R1 Object Module Changes .....</b>	<b>B-12</b>
B.4.1.	\$BANK (Bank Field Changes) .....	B-12
B.4.2.	\$IMPORT (Reference Field Changes) .....	B-12
B.4.3.	\$EXPORT (Definition Field Changes) .....	B-12

## Appendix C. Incompatibilities with ASM

<b>C.1.</b>	<b>Instructions Generated for Overlap Registers .....</b>	<b>C-1</b>
<b>C.2.</b>	<b>Current Location Counter Number .....</b>	<b>C-1</b>
<b>C.3.</b>	<b>Location Counter Local to a Procedure .....</b>	<b>C-1</b>
<b>C.4.</b>	<b>INFO Group 2 .....</b>	<b>C-2</b>
<b>C.5.</b>	<b>NEG Directive .....</b>	<b>C-2</b>
<b>C.6.</b>	<b>SETMIN Directive .....</b>	<b>C-2</b>
<b>C.7.</b>	<b>TYPE Directive .....</b>	<b>C-2</b>
<b>C.8.</b>	<b>Arithmetic — 36 Versus 73 Bits .....</b>	<b>C-2</b>



<b>C.9.</b>	<b>Global Relational Operators</b>	C-3
<b>C.10.</b>	<b>Improved Error Detection</b>	C-3
<b>C.11.</b>	<b>Greater Permissiveness of MASM</b>	C-3
<b>C.12.</b>	<b>Iteration Variables</b>	C-3
<b>C.13.</b>	<b>Propagation of Asterisk Flag</b>	C-4
<b>C.14.</b>	<b>Forms Shorter Than 36 Bits</b>	C-4
<b>C.15.</b>	<b>ASM\$PF vs. MASM\$PF1</b>	C-4
<b>C.16.</b>	<b>Forward References to \$PROC or \$FORM</b>	C-4
<b>C.17.</b>	<b>Directives ON, OFF, and FIELDATA</b>	C-4

## Appendix D. Restrictions, Operational Considerations and Compatibility

<b>D.1.</b>	<b>Restrictions</b>	D-1
<b>D.2.</b>	<b>Operational Considerations</b>	D-1
<b>D.3.</b>	<b>Compatibility</b>	D-1
D.3.1.	MASM System Type Differences	D-1
D.3.2.	Compatability with Previous Release Levels	D-2

<b>Glossary</b>	1
<b>Bibliography</b>	1
<b>Index</b>	1



# Figures

6-1.	Two-Pass Summary .....	6-23
6-2.	One-Pass Summary .....	6-25
6-3.	Words-Given Procedure Summary .....	6-27



# Tables

1-1.	MASM Directives .....	1-4
1-2.	MASM Functions .....	1-7
2-1.	MASM Options .....	2-2
2-2.	Symbolic Input/Output Routine Options (SIR\$) .....	2-4
4-1.	Selectors Defined on the Result of \$BA(e) .....	4-5
4-2.	Truth Tables for Logical Operations .....	4-31
4-3.	The Heirarchy of Operators in MASM .....	4-38
4-4.	Data Type Numbers .....	4-40
4-5.	Description of Type Testing Functions .....	4-41
4-6.	Expression Characteristics Indicator Bits .....	4-41
6-1.	Characteristics of Procedure Types .....	6-21
6-2.	Procedure Types Using Pass-Determination Functions .....	6-30
8-1.	Mode Bit Settings for \$TMODES .....	8-3
8-2.	Expressions Used With \$TMODES .....	8-4
9-1.	Bit Meanings for \$INFO Group 1 .....	9-2
13-1.	MASM Warning Flags .....	13-1
13-2.	MASM Error Flags .....	13-2
A-1.	\$FORCE Relocation Enforcement .....	A-21



# About This Manual

## Purpose

This manual provides programmers and systems analysts with a detailed reference manual for the OS 1100 Meta-Assembler (MASM) processor and language. It includes comprehensive information about the current level of MASM.

This manual is intended to be used as a reference tool by people who are familiar with the MASM language and need specific information about MASM. It is not a tutorial and does not attempt to teach anyone how to write MASM programs.

## Scope

MASM is called a meta-assembler because it is not limited to generating code for a particular hardware architecture. When the MASM processor is loaded, it has the predefined OS 1100 instruction set. However, with the directives and built-in functions provided, the user can alter the environment to generate code for any hardware architecture, if the output of MASM (either relocatable binary format or object module format) can be converted to a form acceptable to the operating system on the alternate architecture.

This manual documents all components and syntax of the MASM software, including input and output formats, controls needed, restrictions, and incompatibilities with OS 1100 Assembler (ASM).

## Audience

This manual is intended to be used by systems and applications programmers who are already familiar with the MASM language and who need specific information about this release level. This manual can also be used by programmers with basic assembler programming knowledge and experience who need to work with MASM.

# Prerequisites

This manual is not a tutorial or a general introduction to MASM programming. This function is left to other texts. Although this manual does not approach MASM on that level, it presents the material in a clear and orderly fashion so that a reader with previous assembler programming experience should be able to learn MASM from it.

# How to Use This Manual

Use this manual as a reference rather than as a tutorial document, to learn the vocabulary, syntax, statements, language characteristics, and program structure of MASM. In addition, this manual contains examples with detailed explanations to reinforce the concepts that are discussed. You can read it through to get familiar with the capabilities of MASM or you can refer to selected pages to learn about one topic in detail.

# Organization

The following list shows the sections and appendixes contained in this manual and briefly describes the contents of each.

## Section 1. Introduction

This section introduces MASM and its general functions and describes the dictionary storage mechanism. The new MASM features for this release are also listed in this section.

## Section 2. MASM Usage

This section describes the following topics:

- The MASM processor call statement and options
- The output produced by MASM
- Input format requirements
- The library searching process

## Section 3. Listing Control

This section describes how to control the printing of output (listings).

## Section 4. Values and Expressions

This section describes the various data types and the syntax and semantics of expressions that evaluate them.

## Section 5. Data Generation

This section describes how MASM generates data. This topic includes formatting and sizing the data, allocating storage, and transforming negative values.



### Section 6. Assembly-Time Controls

This section describes directives that control an assembly at assembly time. This section includes the following topics:

- Directives for handling conditionally assembled code
- Generating repetitive code
- Repeating statement groups
- Transferring control
- Source insertions
- Procedures and functions (creating, accessing, using, and manipulating)
- Microstrings
- Levelers

### Section 7. Dictionary Control

This section describes the functions and general structure of the dictionary and the directives that control it.

### Section 8. Processor Information Functions

This section describes the functions that retrieve the following:

- Date and time information
- The line counter value
- The processor call parameters
- The MASM operating mode settings

### Section 9. MASM Output

This section describes relocatable binary output and use of the \$INFO directive to communicate between MASM and the Collector.

### Section 10. Object Modules

This section describes object module output, the functional replacement for both relocatable and absolute elements. This section contains Unisys restrictions on the support of object modules.

### Section 11. Symbolic Output

This section describes how to create a symbolic element from MASM source output for use as input to other processors.

### Section 12. Saving Definitions

This section describes how to save the results of processing a set of definitions in an omnibus element for subsequent use in other assemblies.

### **Section 13. Error and Warning Diagnostics**

This section defines the warning and error flags generated by MASM.

### **Appendix A. OS 1100 Features**

This appendix identifies the OS 1100 features that are built into MASM.

### **Appendix B. Object Module Evolution**

This appendix lists the changes in the object module environment since release levels 4R1, 4R1A, 4R2, and 5R1.

### **Appendix C. Incompatibilities with ASM**

This appendix explains the incompatibilities between OS 1100 Assembler (ASM) and MASM and indicates how they may be resolved.

### **Appendix D. Restrictions and Compatibility**

This appendix lists the restrictions in using this release of MASM. Compatibility issues are also listed.

## Related Product Information

The following related documents may be helpful when using MASM. Use the version that corresponds to the level of software in use at your site. Consult the *Series 1100 and 2200 Systems Product Information Library Directory* (7831 1578) for specific levels and document numbers.

Unisys is implementing a new documentation numbering system (for example 7830 1234). If a document listed here has a new document number, its previous UP number is also listed.

## Directly Related Documents

The following documents contain information you may need to refer to while using this document.

### ***OS 1100 Exec System Software Executive Requests Programming Reference Manual (7830 7899)***

Previous Document Number: UP-4144

This manual is for system programmers who use Executive requests. It describes the base portion of the Exec operating system and the software needed to write and run user programs.

### ***OS 1100 Collector Programming Reference Manual (7830 9887)***

Previous Document Number: UP-8721

This manual provides the programmer with the necessary documentation to construct executable programs from relocatable elements created by language processors.

### ***OS 1100 Linking System Programming Guide (7831 0521)***

Previous Document Number: UP-9952

Previous Title: OS 1100 New Programming Environment (NPE) Linking System Programming Guide

This guide introduces the Linking System and gives basic static linking and dynamic linking runstreams.

### ***OS 1100 Linking System Programming Reference Manual (7831 0505)***

Previous Document Number: UP-13027

Previous Title: OS 1100 New Programming Environment (NPE) Linking System Programming Reference Manual

This manual discusses advanced Linking System topics and is written for systems and applications programmers who are already familiar with the Linking System guide.

## **Other Related Documents**

***OS 1100 Procedure Definition Processor (PDP) Operations Reference Manual (UP-10070).***

# Notation Conventions

Many examples of MASM code are included throughout this manual where appropriate. These examples are reproduced exactly as they would be printed by the system. In particular, they include a column of numbers in the middle of the code used by the system to identify certain lines.

Each example is followed by a detailed explanation of the lines of code. To facilitate this, we have numbered each line on the left, and refer to these numbers in the explanation. These numbers are different from those in the middle column, which are inserted by MASM.

Change bars in the margins indicate where information has been added or modified for this release of the programming reference manual.

The notation conventions used in the format of statements or clauses and in other portions of this manual are shown in the following table:

Notation	Meaning
A	Capital letters represent entries you must code exactly as shown.
<i>a</i>	Lowercase italic letters represent data you must supply.
[ ]	Items within brackets represent optional entries that you can use or omit.
{ }	Items within braces represent choices from which you select one.
. . .	Ellipses in statement formats indicate entries you can repeat as necessary.
. . . . . .	Ellipses in program examples indicate missing FORTRAN statements deleted because they are not relevant to the example.

# Section 1

## Introduction

MASM (meta-assembler) is not limited to generating code for a particular hardware architecture. When the MASM processor is loaded, it has the predefined OS 1100 instruction set. However, with the built-in directives and functions, you can define the instruction set and useful directives for any hardware architecture, if the output of MASM (OS 1100 relocatable binary or object module format) can be converted to a form acceptable to the operating system on the alternate architecture.

**Note:** *Unisys supports “extended mode” MASM usage (assembler directive `$EXTEND` with or without assembler directive `$OBJ`) only in software written by Unisys or in interfaces written by the customer that explicitly require extended mode assembler-produced elements according to the documentation written by Unisys. In a “nonextended mode” MASM usage (absence of assembler directive `$EXTEND`), Unisys does not support the generation of object modules (use of assembler directive `$OBJ`) but will continue to provide full support for the generation of other provided element types which are not object modules (absence of both the `$EXTEND` and `$OBJ` assembler directives).*

The processor accepts both Fielddata and ASCII input and maintains character constants in either code as specified by the user. MASM uses an internal code to store character constants that do not have to be maintained in a specific character code, such as names in the dictionary and some relocation information.

MASM performs specified tasks based on the interpretation of statements received primarily using the symbolic input/output routine (SIR\$); MASM then produces output that depends on the user's request. The relocatable output routine (ROR) produces a relocatable binary element. The object module output routine (OMOR) produces an object module element. MASM optionally produces a printed listing of the input and its processed form. The structure of the input and output forms is presented in Section 2.

### 1.1. Two-Pass Assembler (Summary and Generative)

MASM performs its function in two scans of the input. The first scan is known as the summary pass, and the second is known as the generative pass. These two passes of the source input, from the first to the last source image, are known as the main assembly. Assemblies called within the main assembly are known as subassemblies. Some initialization is done at the start of each pass (see 6.7.8 and 6.7.1.1).

### 1.2. Dictionary

To use MASM effectively, you must have a general knowledge of the storage mechanism known as the dictionary. The basic function of the dictionary is to store labels and the values associated with them, such as the value and number of the location counter when the label is defined. At processor initialization, the dictionary contains the directives and functions that are built into MASM (see 1.4).

A name and its value are automatically entered into the dictionary when a label is detected on an assembler statement. The value associated with the label is determined by its use in the label field and by the rest of the assembler statement. For example, the value associated with built-in directives and functions is not really a value, but is control information. In this case, the value is not treated as data to be manipulated but as data to control some stage of manipulation.

Each value entered into the dictionary is associated with a type. See 4.4.1 for definitions of the types available.

The dictionary is structured by levels. These levels define the scope of labels and have the range 0 to  $n$ , with 0 being the highest level and  $n$  being the lowest. Labels defined at level 0 are known outside the program. Labels defined at level 1 are known only to the program. Labels defined at levels lower than level 1 are known to selected portions of the program. All operation mnemonics and built-in directives and functions are known at level 1. Dynamic nesting of subassemblies causes lower levels to be used.

When a symbol is presented, MASM searches the dictionary for the value. Value retrieval starts at the level corresponding to the current subassembly and continues by searching progressively higher (lower-numbered) levels until the symbol is found.

## 1.3. New Features and Enhancements

MASM level 6R1 contains the following new features and enhancements:

- A new directive (`$MACH`) to select a desired OS 1100 instruction set.
- A new directive (`$TMACH`) to return the active instruction set.
- A new storage usage statistic (maximum number of words used) added to the MASM end line.
- Two new directives (`$FORCE` and `$FORCEOFF`), used to control a forced relocation assembly.
- The `$EXTEND` and `$PROC` directives have been modified to specify 18-bit addressing.
- A new directive (`$CROSSREF`) to tailor the cross-reference listing.
- A new directive (`$HDG`) to allow the specification of a print heading for the assembly listing.
- The A option on the processor call, to provide additional address checking.
- A new MASM warning flag (B) has been implemented to indicate possible address errors.
- An enhanced relocatable output routine, `ROR$E`, is now used to generate relocatable elements. MASM now requires a Collector level (33R1 or higher) that provides and supports `ROR$E`.

# 1.4. Summary of MASM Directives and Functions

Table 1-1 and Table 1-2 provide a summary of the built-in directives and functions. All directives beginning with \$ have synonyms with the leading \$ dropped (for example, ASCII is a synonym for \$ASCII). Functions, however, do not have synonyms and must be specified with the leading \$ sign (for example, ILCN is not a synonym for \$ILCN).

**Table 1-1. MASM Directives**

<b>Mnemonic</b>	<b>Description</b>	<b>Reference</b>
\$ANDF	An assembly-time directive for conditional construction.	6.1.5
\$ASCII	Sets the system character set to ASCII.	4.2.3.2.1
\$BANK	Defines bank attributes.	10.3.2
\$BASE	Establishes a base register.	A.2.7
\$BASIC	Sets the mode to basic mode.	A.2.3
\$CHAR	Defines a data character set.	4.2.3.2.3
\$CROSSREF	Produces a tailored cross-reference listing.	3.7
\$DEF	Establishes definition mode.	12.2
\$DELETE	Deletes a definition.	7.2
\$DISPLAY	Displays information.	3.3
\$DO	Generates a line repetitively.	6.2.1
\$EJECT	Ejects the page.	3.4
\$ELSE	Part of conditional interpretation.	6.1.2
\$ELSF	Part of conditional interpretation.	6.1.4
\$END	Ends a subassembly.	6.7.1.2
\$ENDD	Ends a \$DO iteration.	6.2.2
\$ENDF	Ends conditional interpretation of a group.	6.1.3

continued



**Table 1-1. MASM Directives (cont.)**

<b>Mnemonic</b>	<b>Description</b>	<b>Reference</b>
\$ENDI	Ends a \$REPEAT iteration.	6.2.5
\$ENDR	Ends a \$REPEAT construction.	6.2.4
\$EQU	Equates a value.	4.1
\$EQUF	Equates a field.	A.2.12
\$EXPORT	Defines the attributes for an exported definition.	10.5.2
\$EXTEND	Sets the mode to extended mode.	A.2.4
\$FDATA	Sets the system character set to Fieldata.	4.2.3.2.2
\$FORCE	Activates forced relocation.	A.2.11
\$FORCEOFF	Deactivates forced relocation	A.2.11
\$FORM	Defines a form.	5.4.1
\$FUNC	Defines a function.	NO TAG
\$GEN	Generates data.	5.3
\$GFORM	A generalized form directive.	5.4.2
\$GO	Transfers control to a \$NAME.	6.4, 6.7.7
\$HDG	Specifies a print heading.	3.8
\$HEX	Sets the binary representation to hexadecimal.	3.6
\$IF	Conditional interpretation.	6.1.1
\$IMPORT	Defines attributes for a reference.	10.4.2
\$INCLUDE	Includes the definitions created using \$DEF.	12.3
\$INFO	Provides special information for the Collector.	9.3
\$INSERT	Inserts images.	6.6
\$LEVEL	Controls dictionary level.	7.5

continued

**Table 1-1. MASM Directives (cont.)**

<b>Mnemonic</b>	<b>Description</b>	<b>Reference</b>
\$LIST	Resumes listing.	3.2
\$LIT	Defines the literal pool.	5.6
\$MACH	Selects a desired OS 1100 instruction set.	A.2.15
\$NAME	Defines an internal name.	6.5, 6.7.7
\$NEG	Defines a function that the assembler calls to transform negative values.	5.8
\$NIL	No action.	6.3
\$OBJ	Produces object module output.	10.1
\$OCTAL	Sets the binary representation to octal.	3.5
\$OUTPUT	Outputs symbolic images.	11.2
\$PROC	Defines a procedure.	6.7.1.1
\$REL	Produces relocatable binary output.	9.2
\$REPEAT	Repeats a statement group.	6.2.3
\$RES	Reserves space in a location counter.	5.5.2
\$SOR	Synonym for \$SYM.	11.1
\$SYM	Establishes symbolic output mode.	11.1
\$UNLIST	Inhibits listing.	3.1
\$USE	Deletes the current base register environment definition and specifies the new base register to be used.	A.2.8
\$WRD	Specifies word size.	5.7

**Table 1-2. MASM Functions**

<b>Mnemonic</b>	<b>Description</b>	<b>Reference</b>
\$	Indicates a change in the number of the active location counter. When used as an expression element, means the same as \$LCV.	5.5.1
\$AP	The numeric part of an integer expression with all relocation information deleted.	4.2.1.2.1
\$BA	A node reference whose elements describe the integer attributes of an integer.	4.2.1.2.2
\$BREG	Returns a value that is the base register associated with an integer expression.	A.2.10
\$CAS	Converts an expression to the ASCII character set.	4.2.3.3.1
\$CB	Converts an integer value to an integer character string.	4.2.3.3.5
\$CD	Converts an integer value to a decimal character string.	4.2.3.3.4
\$CFS	Converts an expression to the Fielddata character set.	4.2.3.3.2
\$CS	Converts an expression to the data character set.	4.2.3.3.3
\$DATE	Returns the date and time in ER DATE\$ format.	8.1
\$FN	Forms a new entry point to a procedure or function.	6.7.10
\$FP	Indicates whether or not this is the final pass of the current subassembly.	6.7.9.1 6.7.9.4
\$GP	Indicates whether or not the current subassembly pass is generative.	6.7.9.2 6.7.9.4
\$HASH	Returns the dictionary class index of an identifier (the system hash value).	7.3
\$IBITS	Returns the indicator bits for an expression.	4.4.3
\$IC	Returns a portion of the MASM symbol dictionary.	7.4

continued

**Table 1-2. MASM Functions (cont.)**

<b>Mnemonic</b>	<b>Description</b>	<b>Reference</b>
\$ILCN	Returns the location counter number in effect at the beginning of the current subassembly pass.	5.5.4
\$LO	Forms a list starting at zero.	4.2.4.3
\$L1	Forms a list starting at one.	4.2.4.4
\$LCB	Returns the address of the first word of the specified location counter.	5.5.6
\$LCFV	Returns the final value of the location counter at the end of the summary pass, without relocation.	5.5.7
\$LCN	Returns the number of the current location counter.	5.5.3
\$LCV	Returns the current value of the specified location counter.	5.5.5
\$LEV	Returns the number of the principal dictionary level.	7.6
\$LF	Returns a description of the waiting label for the specified subassembly.	6.7.5
\$LINES	Returns a count of the number of lines scanned by MASM since the beginning of the assembly.	8.2
\$LP	Indicates whether or not the generative pass of the main assembly is being performed.	6.7.9.3, 6.7.9.4
\$NODE	Forms a node.	4.2.4.5
\$NS	Returns the nth selector.	4.2.4.6
\$PAR	Provides access to the parameters on the MASM processor call statement.	8.3
\$SL	Returns the number of characters in a string.	4.2.3.4.1
\$SN	Returns the selector number.	4.2.4.7
\$SR	Repeats a string a specified number of times.	4.2.3.4.2
\$SS	Extracts a substring from a string.	4.2.3.4.3

continued

**Table 1-2. MASM Functions** (cont.)

<b>Mnemonic</b>	<b>Description</b>	<b>Reference</b>
\$SSS	Substitutes a substring within a string.	4.2.3.4.4
\$TBIN	Type testing for integer.	4.4.2
\$TCON	Type testing for control information.	4.4.2
\$TDAT	Type testing for data.	4.4.2
\$TDIR	Type testing for directives.	4.4.2
\$TFLT	Type testing for floating-point.	4.4.2
\$TFNM	Type testing for a function name.	4.4.2
\$TFUN	Type testing for a built-in function.	4.4.2
\$TINM	Type testing for an internal name.	4.4.2
\$TMACH	Returns the active instruction sets	A.2.16
\$TMODES	Shows what directives are in effect.	8.4
\$TNAM	Type testing for a name.	4.4.2
\$TNOD	Type testing for a node.	4.4.2
\$TPNM	Type testing for a procedure name.	4.4.2
\$TSTR	Type testing for a string.	4.4.2
\$TVAL	Type testing for a value.	4.4.2
\$TYPE	Computes a data type number.	4.4.1
\$XLEV	Returns the dictionary insertion level for new symbols.	7.7



# Section 2

## MASM Usage

This section describes the following topics:

- The MASM processor call statement and options
- The output produced by MASM
- Input format requirements
- The library searching process
- The procedure library search table

### 2.1. Processor Call

MASM is normally a standard processor in SYS\$LIB\$\*MASM of an OS 1100 system. SYS\$LIB\$\*MASM is the standard name recommended for installations. However, MASM can be installed elsewhere based on your site's installation requirements. The *ER Programming Reference Manual* describes the standard form for calling an OS 1100 language or systems processor. This information applies to MASM. Unique to each processor, however, is a set of option letters in addition to the symbolic input/output routine (SIR\$) options.

#### 2.1.1. Processor Options

The MASM processor call statement has the format:

```
@MASM, options    si,ro,so
```

where:

*options*

are defined in Table 2-1 and Table 2-2. MASM assumes the N option if no options are specified in the call statement.

*si*

is the name of the source input element. If omitted, the assumed source is the runstream.

*ro*

is the output element name that depends on the directive used as shown in the following table:

Directive	ro
\$REL (or absence of \$DEF, \$OBJ, or \$SOR)	Relocatable output element name
\$DEF	Omnibus output element name
\$OBJ	Object module output element name
\$SOR	Symbolic output element name

If *ro* is omitted, the assumed name is NAME\$ in TPF\$.

*so*

is the source output element name.

Commas and spaces are significant in the processor call statement since they identify the different components. Therefore, they must be included whenever a preceding entry is omitted.

**Table 2-1. MASM Options**

Option	Description
A	Performs additional address checking to ensure base register specification on extended mode instructions with a base register selector (b-field) and \$EQUFs. A symbol with a \$EQUF must have a base register selector available when used, but it is not required when the symbol is defined.  If no base register is provided by the \$EQUF and B0 is forced on the resulting instruction due to a \$USE directive (only applies to B0 on the \$USE), an A flag is generated since it is assumed that B0 was established as a default and may not be desirable. If you are not using this technique, the A flag can be ignored by turning off the A option after verifying that B0 was the desired resultant base register.
B	Uses batch format in demand mode (assumed if @BRKPT PRINT\$ is active when MASM is initiated).

continued



**Table 2-1. MASM Options (cont.)**

Option	Description
C	Prints source images and \$DISPLAY strings.
D	Prints double-spaced output.
E	Displays error messages and walkback information in case of error, including line numbers and the line that caused the error.
F	Displays warning messages and walkback information in case of warning, including line numbers and the line that caused the warning.
L	Combines S, R, and Y options. The procedure library search table is also printed.
M	Forces MASM to search some of the files in the procedure library search chain before the MASM dictionary (see 2.4.1, 2.4.2). This option allows directive redefinition by procedures (see 6.7.1).
N	Assumes an implied \$UNLIST precedes line 1 of the source language. N is the default option if none are specified on the MASM call line.
O	Prints octal information, including details of the preamble of the output RB element and values produced by \$DISPLAY (other than strings).
R	Prints detailed relocation information for generated data (implies the O option). MASM also provides encoded attribute information and library code name (if attached) in uppercase.
S	Combines C and O options.
T	Reserved.
V	Prints both input and updated line numbers with the correction lines. Valid only in batch runs or with the B option.
X	Terminates the run if errors occur during the assembly (meaningful for batch runs only).
Y	Prints a cross-reference listing immediately after the list of entry points.
Z	Prints a console message if the assembly contains errors.

In addition to the MASM-unique options listed in Table 2-1, the general symbolic input/output routine (SIR\$) options listed in Table 2-2 are available.

**Table 2-2. Symbolic Input/Output Routine Options (SIR\$)**

Option	Description
G*	Input is compressed symbolic in columns 1 to 80 of the symbolic images. Applies only with the I option.
H*	Input contains sequence numbers in columns 73 to 80 of the symbolic images. Applies only with the I option.
I	Reads images from the runstream and inserts them into a new symbolic. If the I option is present without any si specification, SIR\$ reads images from the runstream and passes them to the caller.
J*	Input contains compressed symbolic images in columns 1 to 72 of the images and sequence numbers in columns 73 to 80. These sequence numbers are not checked by the K option. Applies only with the I option.
K*	Checks sequence numbers in columns 73 to 80 of the symbolic images (valid only with H and I options).
P**	Outputs symbolic in Fielddata. (Compare with Q.)
Q**	Outputs symbolic in ASCII. If neither P nor Q is specified, code type of input symbolic is used. If the I option is specified without P or Q, the type of the first call to SIR\$ determines the output type: ASCII, if GETAS\$ call; Fielddata, if GETSR\$ call; and native type if GETNMS\$ call. If both P and Q are specified, output is symbolic with mixed images, Fielddata and ASCII (only if mixed mode code is turned on (NM = 1), otherwise output type is the same as input type).
U	Reads change images from the runstream; applies them to the symbolic input element; and produces a new cycle of the symbolic element.
W	Lists change images.

\* A configurable feature that is disabled in the standard version of MASM.

\*\* Fielddata output was disabled in some earlier SYSLIB versions of SIR\$. Fielddata output is possible again.

### **2.1.2. Reusability**

MASM is a reusable processor. If successive calls on MASM are separated only by transparent control statements (such as @MSG, @LOG, @HDG), MASM is not reloaded from mass storage. Rather, it reads its own control statement, reinitializes itself, and processes the next element. This saves considerable time and I/O resources. This capability is also available when MASM is called from a user file (rather than from SYS\$LIB\$\*MASM or SYS\$\*LIB\$) if all calls on MASM after the first in a sequence do not specify the user file that called MASM (that is, other calls are @MASM...).

To reload MASM, use the @ENDX control statement or the full *qual\*filename.element* processor name. This terminates the reusability sequence.

When MASM is reused, it minimizes the cost of dynamic storage expansion and storage use by starting the next assembly with a storage size the same as the size at the end of the previous assembly.

## 2.2. Output

MASM produces the following types of output that are under the control of the user:

- Printed listing
- Relocatable binary (RB), output module (OM), omnibus element, or symbolic output
- Updated source input

The MASM processor controls the printed listing by allowing text lines to be inserted and new pages started for readability. Insertion of commentary text is easy and encourages clear and complete program documentation. The listing can be partially or completely suppressed.

### 2.2.1. Output Fields

From left to right, the printed output is divided into the following fields:

- Field 1 starts in the first print position and contains error flags, if any.
- Field 2 contains the location counter number and value. It is void unless a relocatable binary element or object module was produced. This field is controlled by the O option.
- Field 3 contains the octal or hexadecimal value associated with the interpretation of field 5. This field is controlled by the O option.
- Field 4 contains source line numbers. It can have several formats.

If two columns of numbers are present, then the left column contains the line numbers of the source output and the right column contains the line numbers of the source input (V option listing).

If only one column of numbers is present, then these numbers are the line numbers of the source output. If the element contains only images produced by the Conversational Time Sharing (CTS) system (which implies no corrections were applied through the symbolic input/output routine), there is a single column of CTS line numbers.

This field is controlled by the C option.

- Field 5 contains the source image seen by MASM. This field is controlled by the C option.

If the element contains only images produced by EDIT 1100 in the Interactive Processing Facility (IPF 1100) environment, there is a single column of sequential line numbers. MASM does not produce segmented line numbers.

### 2.2.2. Preamble Output

At the end of the source input and generated output listing, the MASM processor provides a summary of the preamble of the generated relocatable binary element. This includes the numbers and values of the location counters used, the locations of the externalized symbols, and the names of the undefined symbols.

### 2.2.3. END MASM Line

Following the preamble, the END MASM line contains some statistics concerning processor behavior. These statistics include the following:

- The number of lines processed during the assembly, including the number of lines processed for procedure calls on both assembly passes.
- The assembly time (total accumulated standard unit of processing (SUP) usage) in seconds.
- The storage usage given in the format *a/b/c/d/e*. *a* is the starting size in words of the storage pool. *b* is the maximum number of words used. *c* is the number of storage compactions done. *d* is the number of ER MCODE\$ requests done. *e* is the final size of the storage pool in words. All storage information is in decimal.
- The error and warning summary, a list of all error and warning flags issued and the count of each.

### Example

```
1. @MASM,LEFV source input,source output
2. MASM xxRyy
3. I FLAG GENERATED AT LINE 1
4. I      0 000000 74 06 00 00 0 000000      1.      1      DDDK
5.      -1
6. U FLAG GENERATED AT LINE 2
7. U      000000000000      2.      1+      LAB1      $EQU      XREF
8.      **      BITS 72-0 + XR XREF
9.      -2,2
10.      000001 0000000000024      3.      3      +      20
11.      4.      4      $END      . Comments
12.
13. LOCATION COUNTERS: $(0) 000002  $(1) 000000
14. -----
15. PROCEDURE LIBRARY SEARCH TABLE
16. -----
      .
      .
      .
28. CROSS REFERENCE LISTING
29. ASSEMBLY SYMBOL USAGE X-ETERNAL, L-LABEL, S-SAMPLE/CONDITIONAL LABEL
30. D-DUPLICATED VALUE, N-NODE REFERENCE, P-PROCEDURE, F-FUNCTION, U-UNDEFINED
31. DDDK      1D
32. LAB1      2*
33. XREF      2
34. ASSEMBLY CONTAINS ERRORS: - FLAGS: IU
35. END MASM - LINES: 8  TIME: 2.111  STORAGE: 8028  ERRORS: I(1) /U(1)
```

### Explanation

#### Line 1

MASM call line.

#### Line 2

MASM sign-on line showing the product level, date, and time.

#### Line 3

Error walkback information generated using the E option.

#### Line 4

Shows the breakdown of the listing fields (see 2.2.1).

#### Line 5

Line number generated in field 4 due to the V option on the MASM call line.

#### Line 6

Warning walkback information generated using the F option.

Line 7

The U flag is generated in field 1, and field 3 (controlled by the O option) contains a zero. The four instruction fields are also shown.

Line 8

Relocation information generated, using the R option.

Line 11

A space-period-space construct begins the comment field (see 2.3.1).

Line 13

Location counter information.

Line 15

A table (in search order) of the library files that MASM searched to resolve references (see 2.4.1).

Line 28

The L or Y option produces the cross-reference listing at the end of the assembly.

Line 35

The END MASM line showing diagnostic flags and processor information.

## **2.2.4. Cross-Reference Listing**

When the L or Y option is on the MASM call line, a cross-reference listing is produced. It is derived from the MASM dictionary on the second pass of the assembly and includes label definitions, symbol references, and register usage. The following rules determine if a reference is placed in the cross-reference listing:

- Only references looked for or found at level 0 (external) or level 1 of the main assembly are entered. This includes references in procedures and functions.
- All labels in the source input are included in the cross-reference listing. Labels that are defined at level 0 or 1 of the main assembly are defined with X or L. All other labels (that is, labels defined below level 1 and in unprocessed conditional code) are listed with S.
- Cross-reference is called on the second assembly pass, so tags defined or referenced only on the first pass are not entered. Label definitions from \$INCLUDE elements or procedure definition processor (PDP) procedures are not listed. All labels defined or referenced inside procedures follow the first two rules in this list.
- Symbols encountered while skipping for \$IF or \$GO are not entered. Labels inside skipped conditional code are entered with S.
- Symbols within quoted strings and comment lines (with or without the space-period-space construct) are not listed. MASM directives and instruction mnemonics are not listed.

### Cross-Reference Output Format

Cross-reference output consists of two parts: assembly symbol usage and register usage. Letters are used to define symbol types as follows:

- Label definitions

X

Entry point or external definition. Also known as a label defined at level 0 of the main assembly.

L

Label definition at level 1 of the main assembly.

S

Label in the assembly listing defined below level 1 of the assembly. These labels are usually found in a \$PROC, \$FUNC or \$REPEAT loop. S is also used for labels found in conditional code that is skipped by MASM.

- Symbol references

U

Undefined reference. The symbol was undefined when it was encountered during the assembly.

P

Procedure call.

F

Function call.

N

Node reference or value substitution from a node.

D

Duplicated value; a reference to a label that caused a D flag during the assembly.

, ,

Value substitution (no letter specified).

- Register usage

I

Implicit register references, usually from a \$EQUF reference, listed in the register usage table with I.

The line number for references defined or encountered in procedures or functions reflects the line number of the procedure call. At assembly time, MASM uses sequential line numbers when processing symbols. Thus, a line number specified in the cross-reference listing is always the sequential line number where the symbol was processed, but not necessarily the same line number that appears in the assembly listing.



**Note:** The cross-reference listing can be tailored using the `$CROSSREF` directive (see 3.7).

### Example

```

1. @MASM,LE DOC.CR002,D
2. MASM xxRyy
3.
4. U          10 16 00 00 0 000000
5.    **      BITS 17-0 + XR XREF
6.          00 00 00 01 0 000012
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.          0000000000003
22.
23. D FLAG GENERATED AT LINE 20
24. D          0000000000003
25.    **      BITS 72-0 + LC 0
26. D FLAG GENERATED AT LINE 21
27. D
28.
29.
30.
31. 0 000000 10 00 00 01 0 000012
32. 000001 10 00 00 01 0 000012
33. 000002 10 00 00 01 0 000012
34.
35. 00000000000005
36.
37.
38. D FLAG GENERATED AT LINE 30
39. D
40.
41.
42.
43. 000003 0000000000000
44.
45.
46. LOCATION COUNTERS:    $(0)  000004    $(1)  000000
47.
48. ENTRY POINTS: DOLAB    0000000000006    INLAB    0000000000005
49.
50. PROCEDURE LIBRARY SEARCH TABLE
51.
52. FILE :   INTERNAL :   QUAL*FILE : PDPPROC : @ASGD : FILE IS :
53. # :   NAME :   NAME : SEARCH : BY MASM : @ASGD :
54.
55. 1 : MASM$PF : * : NO : NO : NO :
56. 2 : ASM$PF : * : NO : NO : NO :
57. 3 : SI$$ : GHH *DOC : NO : NO : YES :
58. 4 : PROC$ : SYS$LIB$ *PROC$ : NO : NO : NO :
59. 5 : MASM : SYS$LIB$ *MASM : NO : NO : YES :
60. 6 : SYSLIB : SYS$LIB$ *SYSLIB : YES : NO : YES :
61. 7 : RLIB$ : SYS$ *RLIB$ : YES : NO : YES :
62.
63. CROSS-REFERENCE LISTING
64. ASSEMBLY SYMBOL USAGE X-EXTERNAL LABEL, L-LABEL, S-SAMPLE/CONDITIONAL LABEL
65. D-DUPLICATED VALUE, N-NODE REFERENCE, P-PROCEDURE, F-FUNCTION, U-UNDEFINED
66. ABC 2L
67.
68. ABD 20L
69.
70. AXR$ 1P
71.

```

```

1.          AXR$
2. ABC $EQU + (LA,U A0,XREF)
3. LAB2 $EQUF 10,X1
4.
5. F* $FUNC
6. FLAB* $NAME 2
7. $END
8.
9. P* $PROC
10. PLAB* $EQU 1
11. $END
12. . Skip $IF images
13. $IF 0
14. IFOLAB $EQU 0
15. $ENDF
16. . Process $IF images
17. $IF 1
18. IFLAB $EQU 3
19. $ENDF
20. ABD $EQU START
21.
22. START .
23. I $REPEAT 3
24. REPLAB $EQU 2
25. LA A0, LAB2
26. $ENDR
27.
28.
29. $DO 2 ,DOLAB* $EQU 6 .
30.
31. START .
32.
33. P
34. +F
35. END

```

## MASM Usage

---

```
72. DOLAB          29X
73.
74. F 5L          34F
75.
76. FLAB          6L
77.
78. I             25L
79.
80. IFLAB         18L
81.
82. IFOLAB        14S
83.
84. INLAB         27X
85.
86. LAB2 3L       25
87.
88. P             9L   33P
89.
90. PLAB          10S  33L
91.
92. REPLAB        23S  25L
93.
94. START         20D  21L  30L
95.
96. XREF          2U
97.
98. REGISTER USAGE - UNISYS 1100 SERIES (USER REGISTERS)
99.   I = IMPLICIT REGISTER USAGE VIA EQU
100. X1            3   25I
101.
102. A0            2   25
102. ASSEMBLY CONTAINS ERRORS:      -   FLAGS: DU
103. END MASM - LINES: 188   TIME: 1.561   STORAGE: 28950   ERRORS: D(3) /U(1)
```

### Explanation

The following table clarifies the listing example:

Line	Cross-reference	Explanation
70	AXR\$	Procedure call at line 1 of the assembly.
74	F	Level 1 label definition at line 5; function call at line 34.
82	IFOLAB	Label found inside unassembled conditional code.
86	LAB2	Level 1 label definition at line 3; reference at line 25 within \$REPEAT.
90	PLAB	Label found in sample code at line 10; level 1 label definition at line 33 within procedure P.
94	START	Value reference at line 20 that caused a D flag; level 1 label definition at lines 21 and 30.
96	XREF	Undefined reference at line 2.
100	X1	Register X1 used at line 3 and implicitly used through \$EQUF LAB2 at line 25. Line 25 (the end of the \$REPEAT loop) is the line where words from the \$REPEAT loop are generated.
102	A0	Register A0 referenced at lines 2 and 25.

## 2.3. Input

This section defines the MASM input format, requirements, and rules.

### 2.3.1. Statements

MASM processes the input presented to it by statements, where a statement is one or more lines of input text and a line is an 80-character image. A statement has the following parts:

- The *functional* part that is interpreted by MASM
- The *comment* part that provides additional information to the reader (see 2.3.3.4)

The functional part of a statement has the following fields:

- Label field (see 2.3.3.1)
- Operation field (see 2.3.3.2)
- Operand field (see 2.3.3.3)

Each field can contain subfields. All of the fields and subfields following the label field are in free form. The label field must begin in column 1 of the symbolic line. Any or all of the fields can be void. Fields are generally bounded by one or more spaces; subfields are bounded by commas.

MASM completes the interpretation of the functional part of a line when it encounters one of the following:

- The maximum number of fields and subfields required by the operation
- The 80th character
- The line terminator space-period-space ( . )
- A line continuation character ( ; ) that is not included in a string enclosed in quotation marks

#### Example 1

```
PF      $FORM      12,6,18      . FORM DEFINITION
```

This example uses all four fields. PF is the label; FORM is the operation field; 12,6,18 are three subfields in the operand field; characters to the right of the period make up the comment.

### Example 2

```
. A COMMENT
```

This line contains only the comment field. It could indicate a logical break in the symbolic code or give additional commentary.

### Example 3

```
$END
```

This line contains only an operation field. MASM recognizes this as an operation field because it is the first field not starting in column 1.

## 2.3.2. Symbols

Labels, operations, and many operands are generally specified as strings of characters, called symbols, that are subject to certain restrictions. These restrictions eliminate ambiguities and protect the processor.

Legitimate MASM symbols can contain the characters A-Z, 0-9, \_, and \$. A symbol may not begin with a digit or underscore, and only MASM system symbols may begin with the \$ character. Therefore, all user-defined symbols must begin with an alphabetic character. If an element is being maintained in ASCII, no distinction is made between uppercase and lowercase characters when used in symbol names. Thus, ABC and abc are the same symbol. The maximum length of a symbol is 12 characters. Any characters beyond 12 are ignored.

**Note:** *The Fielddata underscore (077) is the stop character for demand terminals. A Fielddata line of output from the underscore character onward is not displayed.*

### Example 1

```
AB1
```

This is a valid MASM symbol.

### Example 2

```
1AB
```

This is an invalid MASM symbol.

### Example 3

```
A#7
```

This is an invalid MASM symbol.

### Example 4

`$EQU`

This is an invalid symbol if you try to insert it into the dictionary. It is a valid symbol if you are referring to the MASM directive `$EQU`.

### Example 5

`A/B`

This is an invalid symbol, but it is a valid expression.

### Example 6

`AVERYLONGNAME`

This is a valid MASM symbol, but is the same as `AVERYLONGNAM` because MASM accepts only 12 characters.

## 2.3.3. Fields

The functional part of a MASM statement consists of the following fields:

- Label
- Operation
- Operand

### 2.3.3.1. Label Field

This field is optional and is used to introduce symbols into the dictionary. The label field can be divided into subfields. These subfields can be further divided into items. The following entries are allowed as items, in the order given:

#### 1. Page control

The first item can be a slash (/) to indicate page eject.

#### 2. Line levelers

The next item on the line can be a line leveler in the form `%n`, where *n* is an unsigned integer (see 6.9).

#### 3. Location counter specifications

The leveler can be followed by a location counter specification in the form `$(e)`, where *e* is an expression evaluated as an integer in the range of 0 to 63.

### 4. MASM symbols

The next item in the subfield can be a legal MASM symbol. If both a location counter specification and a symbol appear in the label field, they must be in separate subfields, (that is, separated by a comma). One or more dictionary control items (identified by \*) can follow the MASM symbol (see 2.3.3.1.1).

### 5. Node selectors

The next item can be a node selector in the form  $(e_1, e_2, \dots)$ , where  $e$  indicates some expression that can be converted to an integer.

### 6. Dictionary insertion specifications

The dictionary control items can also appear after the node selectors if there were none before the node selectors; these items cannot appear in both places. In special cases, the label field can consist of one or more asterisks. (See 6.7.4 for a discussion of waiting labels.)

There can be more than one item within a legal MASM symbol. The symbols are set to the value of the current location counter. If the statement has a directive that uses a label in its interpretation, the last MASM symbol in the label field is associated with the directive.

#### Example 1

```
/
```

The label field consists of a single item, the slash (/). The slash causes a page eject.

#### Example 2

```
/ABC
```

The label field consists of a page eject character (/) and a symbol. The symbol is implicitly defined.

#### Example 3

```
%1:$(3),TAG
```

The label field consists of two subfields. The first subfield contains the following items:

- A line leveler
- A location counter specification.

The second subfield consists of a symbol.

### Example 4

```
EOFADR*
```

The label field consists of a symbol and a dictionary control character (\*).

### Example 5

```
ARG*(1,5)  
ARG(1,5)*
```

The label fields of both these lines have the same effect. The label field of both lines consists of the following items:

- A symbol
- A node selection
- A dictionary control character

### Example 6

```
TAG,I0G
```

The label field consists of two subfields, each containing a symbol. The value assigned to each symbol depends upon its use.

### Example 7

```
TAG,K      $DO      10,+K
```

The symbol TAG will be assigned the value of the current location counter. The symbol K is assigned incrementing values (1, 2, ...). (See 6.2.1 for a discussion of \$DO.)

#### 2.3.3.1.1. Externalized Labels

A label can be externalized (known outside of the program). It is externalized when placed at level 0 of the dictionary. MASM inserts labels in the main assembly into level 1 of the dictionary. Each asterisk (\*) suffixed to a MASM label inserts that label one level higher in the dictionary.

For example, assume a processing level of 1 for the following line:

```
TAG*      $EQU      6
```

When MASM encounters this line, it inserts the symbol TAG at level 0 of the dictionary. This causes the symbol to be known outside the program. Such labels are entered into the entry point table in the preamble of the relocatable binary element.



In the object module environment, labels should be externalized by the \$EXPORT directive. If you do not use the \$EXPORT directive, MASM attaches default attributes to labels that are externalized in level 0 of the dictionary.

The object module environment is entered when you use the \$OBJ directive. MASM interacts with the Linking System to provide the externalized label as a definition within the object module element. For more information on definitions and the \$EXPORT directive, refer to Section 10.

### 2.3.3.1.2. Node Selectors

MASM permits use of nodes and selectors as labels. A selector can be any legitimate assembler item, expression, or another node and selector. A label cannot be used as its own selector. (See 4.2.4 for more information.)

A selector is enclosed in parentheses and follows the symbol immediately, with no intervening spaces.

#### Example 1

```
A(4)
```

This selector is a single integer.

#### Example 2

```
A(3,2)
```

This is a multiple selector that identifies a multiple dimension element.

#### Example 3

```
X(1,Y(1))
```

This selector contains a variable.

#### Example 4

```
X(4,3,SIZE//2)
```

Selectors can be arbitrary expressions.

### 2.3.3.2. Operation Field

The operation field starts with the first nonblank character following the label field and terminates with a blank. The first subfield of an operation field must be a MASM directive, procedure reference, function reference, or instruction. Otherwise, the operation field is considered void. Subsequent subfields act as operand input for the operation specified.

### Example 1

```
$RES      5
```

The operation field consists of one subfield (\$RES) and is a valid MASM directive.

### Example 2

```
A      $EQU      2
```

The operation field consists of one subfield (\$EQU) and is a valid MASM directive.

### Example 3

```
APROC,1,2
```

The operation field consists of three subfields. Subfield 1 is a user-defined procedure. Subfields 2 and 3 are objects which the procedure can refer to.

### Example 4

```
LA,U
```

The operation field consists of two subfields. Subfield 1 may be an instruction mnemonic. Subfield 2 is operand information to be used when generating the instruction.

### Example 5

```
+      14
```

The operation field is void.

## 2.3.3.3. Operand Field

The operand field of a MASM statement can contain multiple fields. It is part of the functional portion of a MASM statement.

The operand begins with the first nonblank character after the operation field (or label field if the operation field is a void) and continues until the end of the functional portion of the statement. The operand can consist of more than one field, or it can be a void.

It is not necessary for the operand field to contain the maximum number of subfields implied by the operation field. When omitting a subfield other than the normal first field or last field, the construct comma-zero-comma (,0,) or two contiguous commas (,,) is necessary. If the last subfield is omitted, a comma is not required after the last coded subfield.

Any subfield referenced but not specified in the operand is evaluated to zero.

Any subfield of any field of the operand portion can be flagged by prefixing the subfield with an asterisk (\*). An asterisk cannot stand alone in a subfield, although \*0 is acceptable.

### Example 1

```
A      $EQU      2
```

The operand consists of a single field, the value 2.

### Example 2

```
APROC      ABC,*0
```

The operand consists of a single field with two subfields. The first subfield is the symbol ABC. The second subfield is a flagged item with the value zero.

### Example 3

```
LA,U      A0, TAG
```

The operand consists of a single field with two subfields. The first subfield is the symbol A0, and the second subfield is the symbol TAG. The leading space in the second subfield is ignored.

### Example 4

```
APROC      APPLE TREE
```

The operand consists of two fields. Field 1 is the symbol APPLE, and field 2 is the symbol TREE.

### Example 5

```
APROC      APPLE,,02
```

The operand consists of a single field with the following subfields:

- The symbol APPLE
- A void subfield that evaluates to zero
- The value two

### Example 6

```
$ASCII      A COMMENT$
```

ASCII is a MASM directive that does not require any operand fields. The functional portion of the statement ends with the space following the symbol \$ASCII. The space-period-space ( . ) construct is recommended for specifying comments (see 2.3.3.4).

### 2.3.3.4. Comments Field

The comment field is that part of a MASM statement not containing the functional part.

The construct space-period-space ( . ) marks the end of the functional portion of a MASM statement and the beginning of a comment. If the functional portion is redefined and MASM needs to look for more data (without the space-period-space construct), it takes data from the comment field. All characters are allowed in the comment field.

### Example 1

```
APROC1      COMMENT PART OF STATEMENT
```

APROC1 is a user-defined procedure that does not require any operand fields. Therefore, the functional portion of the statement ends with the space following the symbol APROC1.

### Example 2

```
APROC2      APPLE;      A COMMENT  
            TREE        ANOTHER COMMENT
```

APROC2 is a user-defined procedure that requires two fields. The line continuation (;) marks the end of the functional portion of the first line. The second line contains the continuation of the functional part of the first line. TREE is the second field of the procedure reference. The rest of the line is a comment.

### Example 3

```
APROC4      APPLE . A COMMENT MAY GO HERE
```

APROC4 is a user-defined procedure that might or might not reference more than one field. The construct space-period-space marks the end of the functional portion of the MASM statement. Any references to fields beyond APPLE evaluate to zero.

## 2.3.4. Line Continuation

The line continuation character is the semicolon (;). If a MASM statement is longer than 80 characters, use a semicolon to continue the statement onto the next line. Text on the continuation line can begin in column 1.

When MASM encounters a semicolon outside a quoted string, scanning of the functional part of the line terminates and the remainder of the line is the comment part. MASM assumes the functional part of the next statement begins with the first nonblank character of the next line.

There is no limit to the number of continuation lines. However, readability should be considered in complex statement structures.

### Example 1

```

1          TAG          $RES ;          A COMMENT
2                                15          ANOTHER COMMENT
3          TAG ;
4          $RES ;
5                                15
```

### Explanation 1

Lines 1 and 2 produce the same results as lines 3, 4, and 5.

To continue a quoted string, terminate the string of the current line with a single quotation mark, immediately followed by a semicolon. If the first nonblank character on the continuation line is a single quotation mark, and the string on the continuation line is a valid MASM quoted string, the two strings are concatenated.

### Example 2

```

1          'ABCD';          THE COMMENT PORTION
2          'EFG'
```

### Explanation 2

Lines 1 and 2 produce the string 'ABCDEFGF'

### 2.4. Library Searching

Directive type symbols being processed by MASM are resolved in the MASM dictionary or in certain library files. The files that MASM searches for these symbols make up the procedure library search chain.

The assembler procedure table of these library files is searched for a specific symbol. If the symbol is found, the definition or sample is read in from the file and placed in the MASM dictionary, and the search for that symbol ends.

If the M option is on the MASM call line, part of the library search chain is searched for symbols before the MASM dictionary is searched (see 2.4.2). This allows you to redefine all instruction mnemonics and MASM directives without a leading \$ that reside in the MASM dictionary.

**Note:** *Even though this capability exists, it is not recommended that the instruction mnemonics and MASM directives be redefined. Use of the M option increases assembly time to perform the additional searching. Also, other products or applications that can use the same library files and the M option can be adversely affected by these redefinitions.*

If the M option is not set, the MASM dictionary is searched before any of the library files are searched.

If the symbol is found at any stage of the search, the search ends and the definition or sample is read from the file or MASM dictionary where it was found. If the definition was found in the assembler procedure table of a library file, the definition is placed in the dictionary. File searching for the symbol does not occur again unless the definition is deleted, using the \$DELETE directive.

A table of the procedure library files searched can be listed at the end of the MASM assembly. (See NO TAG).

**Note:** *The files searched in the procedure library search chain can be configured so that additional MASM\$PF files are searched or system files are removed. Refer to the generation information for MASM level 6R1 that is on the release tape.*

## 2.4.1. Procedure Library Search Chain

The procedure library file search chain is as follows:

```
MASM$PF
MASM$PF1 or ASM$PF
.
.
.
MASM$PFx (where x = LIBMAX)
source input(SI$$), source output(SO$$), resultant output(RO$$)
PROC$ or SYS$LIB$*PROC$
MASM or SYS$LIB$*MASM
SYSLIB or SYS$LIB$*SYSLIB
RLIB$ or SYS$*RLIB$
```

Only one file on each line is searched. The file(s) to the right of the first file on the line is the alternate file to search. If none of the files on a line is available, that place in the search order is skipped.

## 2.4.2. Search Order

The sequence of directive type symbol lookup is as follows:

1. If the M option on the MASM processor call statement is not set, the dictionary is searched.
2. The MASM\$PF files are searched. The number of files searched depends on how many are configured and assigned to the run (see 2.4). The default configured files are MASM\$PF and MASM\$PF1.

**Note:** *The file name MASM\$PF1 replaces the file name ASM\$PF. For compatibility, MASM recognizes an ASM\$PF file if the ASM\$PF file is attached to a file assigned to the run and if the file name MASM\$PF1 is not attached to a file assigned to the run.*

3. If there is a source input file, it is searched. If not, the source output file is searched. If none, the relocatable output file is searched. Only one of these files is searched in this position.
4. If the M option is on the MASM processor call, the dictionary is searched. The dictionary is searched only once, either first (no M option) or here.
5. The system files are searched. These files, in search order, are: PROC\$, MASM, SYSLIB, and RLIB\$. MASM can be configured to remove some or all of these system files from the search (see 2.4).
  - a. If any file named qualifier\*name (where name is one of the above system file names) is assigned to the run, the file is searched. If the @USE name of name is attached to a file assigned to the run, the file is searched. If more than one file is assigned (using file name or @USE name) with a specific system file name, the operating system determines which file is searched.

- b. If the system file name is not assigned, and name is not attached (using @USE) to another file, MASM attempts to assign, search, and free a default system file. The default system file for PROC\$, MASM, and SYSLIB is SYS\$LIB\$\*name. SYS\$\*RLIB\$ is the default system file for RLIB\$.

If a specified find is made at any stage of this search, the search ends and the definition or sample is read in from the file where it was found. Once the definition or sample is found and read in from a file, its definition is placed in the dictionary and file searching for the symbol does not occur again unless the definition is deleted. Using alternate files (see 5a) is generally not recommended. If you do not want MASM to keep assigning and freeing these files, you can assign the default files SYS\$LIB\$\*SYSLIB and SYS\$\*RLIB\$.

**Note:** *If MASM assigns these default files, MASM will not free them when it is reusable until MASM reusability is terminated normally.*

Files in the latter stages of the search order can be searched earlier if you have a file (or files) that meets the criteria for an earlier stage in the search order. For example, the file name MASM\$PF1 is attached to SYS\$\*RLIB\$ which is already assigned to the run. You can do this intentionally to ensure that a file that is normally searched later will be searched earlier. Thus, a definition that can occur in more than one search file can be found in the desired search file, depending on how you have adjusted the search order.

**Notes:**

1. *If a search file is moved up in the search order by meeting the criteria for an earlier search order stage, it is also searched in any other stage of the search order whose criteria it meets. This includes its normal stage if the definition being searched for has not been found.*
2. *In a COMUS or SOLAR installation environment, library elements that formerly resided in SYS\$\*RLIB\$ are now in other files. MASM\$PF and MASM\$PF1 can be attached as @USE names to the other files. These files should be assigned to access the library elements. This requires the user to know which files contain these elements and the order in which the files should be searched in some cases.*
3. *MASM searches SYS\$LIB\$\*RLIB\$ only when you instruct it to do so. SYS\$LIB\$\*RLIB\$ is a migration file that allows you to move from an Exec environment lower than level 39 to the library structure associated with an alternate file common bank (AFCB) environment.*



## 2.5. Procedure Library Search Table

The procedure library search table (PLIBT) lists the search order of the library files in the procedure library search chain. The PLIBT table is printed if the L or Y option is on the MASM call. In the absence of an L or Y option, the PLIBT table is printed only when in batch mode (see the B option in Table 2-1), and either the S and E or the S and F options are on the call line.

In addition to listing the search order for all MASM\$PF and system files, the following additional information is provided in the table for each search file:

PDPPROC SEARCH (Column 4)

If a file corresponding to the internal file name was assigned to the run and has an assembler procedure table, the flag is set (YES) and the external *qual\*file* name is listed.

@ASGD BY MASM (Column 5)

This field is YES if MASM assigned the file. MASM attempts to assign only default system files.

FILE IS @ASGD (Column 6)

This field is Yes if the file was assigned to the run. This flag also indicates if \$INCLUDE elements can be searched for in this file when the element is specified without a *qual\*file* name (see 12.3).

### Example

The following is an example of PLIBT table output:

PROCEDURE LIBRARY SEARCH TABLE							
FILE #	INTERNAL NAME	QUAL*FILENAME	PDPPROC SEARCH	@ASGD BY MASM	FILE IS @ASGD		
1	MASM\$PF	GHH *UTIL	YES	NO	YES		
2	ASM\$PF	GHH *JUNK	NO	NO	NO		
3	RO\$	GHH *TPF\$	YES	NO	YES		
4	PROC\$	SYS\$LIB\$ *PROC\$	NO	NO	NO		
5	MASM	SYS\$LIB\$ *MASM	NO	YES	YES		
6	SYSLIB	SYSTMP *REL\$	NO	NO	NO		
7	RLIB\$	SYS\$ *RLIB\$	YES	YES	YES		

### Explanation

The following explanations are by file sequence number (FILE #).

1. The file GHH\*UTIL has the @USE name of MASM\$PF attached, is assigned to the run, and has an assembler procedure table.
2. The file or @USE name MASM\$PF1 did not exist. The alternate search name ASM\$PF is attached to the file GHH\*JUNK. The file is either not assigned to the run or does not exist. This position in the search order is skipped.
3. The relocatable output file GHH\*TPF\$, taken from the MASM call line, was assigned to the run and had an assembler procedure table. One file from source input, source output, or relocatable output is picked to search in this position.
4. The file or @USE name PROC\$ was not assigned to the run. MASM was unable to assign the default SYSTEM file SYS\$LIB\$\*PROC\$. This search position is skipped.
5. The file or @USE name MASM was not assigned to the run. MASM assigned the file SYS\$LIB\$\*MASM to the run. The file had no assembler procedure table and was not searched for PDP procedures. This file can be searched for \$INCLUDE elements since it is assigned to the run.
6. The @USE name of SYSLIB was attached to the @ASGd file SYSTMP\*REL\$\$ to be searched in this position.
7. The file or @USE name RLIB\$ was not assigned to the run. MASM successfully assigned the default file SYS\$\*RLIB\$ to search in this position.

# Section 3

## Listing Control

This section describes how to control the printing of output (listings).

### 3.1. \$UNLIST — Inhibit Listing

The \$UNLIST directive requires no parameters. MASM ignores this directive if the current subassembly does not generate output. Otherwise, MASM inhibits any listing until it encounters a \$LIST directive. The N option on the processor call places an \$UNLIST directive before the first line of the assembly. This is not the same as omitting all listing request options, since a \$LIST directive resumes listing. To ensure no listing, do not specify any of the listing options (C, D, E, F, L, O, R, or S).

### 3.2. \$LIST — Resume Listing

The \$LIST directive requires no parameters. MASM ignores it if the subassembly pass does not generate output. Otherwise, MASM removes any UNLIST condition in existence (due to an initial N option or an \$UNLIST directive) and resumes the printed listing controlled by listing options specified on the MASM processor call statement.

### 3.3. \$DISPLAY — Display Information

The format of the \$DISPLAY directive is as follows:

`$DISPLAY[e0] e1,e2,...,en`

where:

*e*<sub>0</sub>

is an integer.

*e*<sub>1</sub>,...,*e*<sub>*n*</sub>

are integer values or strings in the system character set.

An *e*<sub>0</sub> parameter equal to 1 is optional on the \$DISPLAY directive. An *e*<sub>0</sub> value of 1 causes printing, even if an UNLIST directive is on. However, even with *e*<sub>0</sub> set, no printing occurs if the MASM call line does not have a print option.

## Listing Control

---

Use the `$DISPLAY` directive with an optional  $e_0$  parameter of 1 as follows:

```
$DISPLAY,1  e1,e2,...,en
```

MASM ignores  $e_1, \dots, e_n$  if the current subassembly pass does not generate output. Otherwise, the strings and integer values appear in the printed listing (according to option specification).

Strings print in the position normally occupied by the source image; however, no line numbers appear. A flagged string produces an E flag.

An integer value prints in the position normally occupied by the assembler output; however, no location counter is specified. An integer value always prints as soon as encountered; but a string prints only when another string is encountered in the parameter list, an integer value is printed, or the end of the parameter list is reached.

You can use the `$DISPLAY` directive to provide error indication messages from inside procedures and to document assembly-time actions, such as large table generations that are otherwise unreadable. Strings can be composed dynamically by using concatenation operators.

### Example 1

Assume an integer value `V` and the following statements:

```
1. $DISPLAY      'V' 'V'
2. $DISPLAY      V, 'V'
3. $DISPLAY      *'ERROR', V
```

### Explanation 1

Line 1 displays the information on one line. Line 2 requires two lines for display. Line 3 produces an E flag and displays `ERROR` and the value of `V` on the same line.

### Example 2

```
1. @MASM,NSE symbolic-input,.symbolic-output
2. $DISPLAY,1 5,'ABCD'
3. $DISPLAY '3','STRING'
4. $LIST
5. $DISPLAY 7,'THIS IS IT'
6. $END
```

### Explanation 2

Line 1 turns on the `UNLIST` directive (with an `N` option) and also sets the `S` and `E` option. Line 2 displays an integer value 5 and the string `ABCD`. Line 3 displays nothing. Line 4 resumes the listing. Line 5 prints the source image and displays the integer value 7 and the string `THIS IS IT`.

### 3.4. Page Ejection

A slash (/) appearing in column 1 (see 2.3.3.1) advances the printing to the top of the next page. The slash appears on the new page.

The \$EJECT directive requires no parameters. If the current subassembly pass is not generative, this directive is ignored. Otherwise, the paper is advanced so that the following printing begins on a new page.

### 3.5. \$OCTAL — Set Binary Representation to Octal

The \$OCTAL directive requires no parameters. It sets the binary representation mode to octal.

### 3.6. \$HEX — Set Binary Representation to Hexadecimal

The \$HEX directive requires no parameters and sets the binary representation mode to hexadecimal.

### 3.7. \$CROSSREF — Tailored Cross-Reference Listing

The format of the \$CROSSREF directive is as follows:

```
$CROSSREF[,  $e_0$ ] [ $e_1$ [,  $e_2$ [,  $e_3$ ]]]
```

where:

$e_0, e_1, e_2, e_3$

are nonrelocatable integer expressions between zero and 1.

\$CROSSREF is used to control the output of the cross-reference listing. If the value of the expression  $e_0$  is equal to 1 then the cross-reference listing will be single-spaced. Single-spacing means that no blank line will appear between different cross-reference items.

If the value of the expression  $e_1$  is equal to 1, then the cross-reference listing will not contain any register usage items. This includes both implicitly and explicitly used registers.

If the value of the expression  $e_2$  is equal to 1, then the cross-reference listing will not contain any labels defined below level 1 of the dictionary, such as those defined in \$PROCs, \$FUNCS, and \$REPEATs. This also will inhibit the listing of items related to conditional code that was skipped by the assembler.

If the value of the expression  $e_3$  is equal to 1, then the cross-reference listing will be disabled for the remainder of the assembly or until another \$CROSSREF directive is encountered. If the value of the expression  $e_3$  is not equal to 1, the \$CROSSREF directive will again enable the cross-reference listing.

If \$CROSSREF is used with no  $e_0$ ,  $e_1$ ,  $e_2$  or  $e_3$  specified, the result is a normal cross-reference listing. This is the default case when the cross-reference listing is being generated.

## 3.8. \$HDG — Specify Print Heading

The \$HDG directive is used to specify a print heading (or title) for the assembly listing. The format of the \$HDG directive is as follows:

```
$HDG[ ,  $e_0$ ]   $e_1$ [ ,  $e_2$ ]
```

where:

$e_0$

is a nonrelocatable integer expression between 0 and 1.

$e_1$

is a nonrelocatable string in a system character set.

$e_2$

is a nonrelocatable integer expression between 1 and 3.

If the value of the expression  $e_0$  is equal to 1, the \$HDG directive performs a page eject after submitting the print heading, using an ER PRTCN\$ (or ER APRTCN\$).

If the value of the expression  $e_0$  is equal to 0, no eject will be generated. The new heading can be displayed by requesting a page eject, using either a slash (/) character or \$EJECT. Otherwise, the new heading will not be displayed until a normal page eject occurs. If multiple \$HDG directives are specified, the last one processed at the time of the page eject will be used.

Parameter  $e_1$  is a string constant that is submitted on the ER PRTCN\$ (or ER APRTCN\$) as the new print heading.

**Note:** *The string constant has a maximum length of 96 characters and cannot contain any periods.*

## Listing Control

---

Parameter expression  $e_2$  controls the options submitted on the heading print control function (@HDG) as described in the following table:

Value	Heading Print Control Result
1	The N-option will be placed on the heading function, suppressing the printing of the heading, date, and page number (@HDG,N).
2	The page number will be set to 1 for the heading function. The page numbering will start at 1 instead of the print file's current page number (@HDG,P).
3	The X-option will be placed on the heading function, suppressing the printing of the date and page number (@HDG,X).

**Note:** *The \$HDG directive is only meaningful in conjunction with the @BRKPT statement when you are directing output into a user-defined file. In a batch run, headings go directly into PRINT\$ with no need of the @BRKPT statement.*



## 3.9. Listing Control Example

The following example uses some of the MASM listing control directives.

### Example

```
1. @MASM,S MASM*MSM4.LI001,TPF$.
2. MASM xxRyy
3.
4. 0 000000 0000000000012
5.
6. 000001 00000000A
7.
8. 000002 0000000000012
9.
10.
11. LOCATION COUNTERS: $(0) 000003 $(1) 000000
12. END MASM - LINES: 14 TIME: 4.120 STORAGE: 8014
```

1.	\$eject	
2.		+10
3.	\$hex	
4.		+10
5.	\$octal	
6.		+10
7.	\$end	

### Explanation

#### Line 4

The default binary representation is octal.

#### Line 5

The \$HEX directive sets the binary representation mode to hexadecimal.

#### Line 7

The \$OCTAL directive sets the binary representation mode to octal.



# Section 4

## Values and Expressions

Values are the fundamental elements of MASM subfields. Values are computed by the evaluation of expressions. Values can be retained by assigning the value to symbols or node selectors. MASM has several data types and allows explicit use of typing. This section describes the various data types and the syntax and semantics of expressions that evaluate these types.

### 4.1. \$EQU — Equate a Value

Call the \$EQU directive as follows:

```
label      $EQU      e
```

where *e* is an expression. If the label is absent, no action is taken. Otherwise, the expression *e* is converted according to the rules for parameter conversion. When the expression contains an undefined identifier, MASM searches the dictionary to see if the identifier is defined as a directive, an internal name, or a procedure name. If such a definition is found, it is taken as the value of the expression. The label is then given the value of the conversion as its definition. If no such definition is found, MASM converts the identifier to an integer value with relocation. (See 4.2.1.2.)

The following statements have the same result of equating the tag MACRO to the directive PROC. The second form is preferred since it is clearer.

```
MACRO      $EQU      PROC  
MACRO      $EQU      /PROC
```

MACRO can now be used as a synonym for the \$PROC directive. (See 4.3.1.5.4 for information about the slash (/) control information operator.)

Symbols given definitions by use of the \$EQU directive are known as explicit definitions and can be redefined without generation of a D flag.

## 4.2. Values

A knowledge of the various kinds of values (also referred to as data types) is necessary to use the full power of MASM for constructing programs. Extremely general procedures can be constructed, which base their operation on the nature of the data submitted as parameters. A full set of built-in functions is available for transferring one value to another, and for testing the value of a parameter. These are described in 4.4.

### 4.2.1. Integer Values

The internal arithmetic precision of MASM for integer arithmetic is 72 bits plus a sign. MASM performs all integer computations in this 73-bit arithmetic, including those that generate a single-precision word on output.

Integers can be specified in decimal, octal, or hexadecimal. The interpretation of a number as octal or hexadecimal depends on the setting of a global assembly switch, controlled by the directives `$OCTAL` and `$HEX`.

Decimal numbers must begin with the digit 1, 2, 3, 4, 5, 6, 7, 8, or 9; cannot begin with 0; and cannot contain a decimal point. Any of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9 can be used in a decimal number.

Octal or hexadecimal numbers must begin with 0. If the current mode is octal, only the digits 0, 1, 2, 3, 4, 5, 6, and 7 are permitted. If the mode is hexadecimal, the digits permitted are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. If the last character of a hexadecimal constant is D, it is interpreted as a digit, not the double-precision postfix operator. Therefore, place parentheses around the hexadecimal constant with the D suffixed to the right parenthesis to generate a double-precision hexadecimal constant; that is, (054D)D.

Another integer item is the label reference. It retrieves a value computed earlier by MASM and assigned to the label by the `$EQU` directive, the `$EQUF` directive, or implicit definition. An implicit definition means the label appeared in the label field of an instruction or other generated line. Labels can be relocatable relative to internal location counters of the element being assembled or relative to external symbols. Two relocatable values are not considered equal unless both the absolute part and the relocation are the same. Integer values can also have a format attached (`$FORM` directive, see 5.4.1). A format defines the layout of fields within a word for an item. To create such a format, use the `I$` or `EI$` built-in format (see A.2.8), the `$EQUF` directive (see A.2.8), an instruction, or a programmer-defined format (created with the `$FORM` directive).

## Example

```

1. @MASM,S MASM*MSM4.VA001,TPF$.
2. MASM xxRyy
3.
4. 0 000000 0000000000016
5. 000001 0000000000014
6.
7. 000002 0000000018
8. 000003 000000000F
9.
10.
11. U 1 000000 08 0 0 0 0 0000
12.
13.
14. LOCATION COUNTERS: $(0) 000004 $(1) 000001
15.
16. EXTERNAL REFERENCES: ABC14 A0
17.
18. ENTRY POINTS: START $(1) +000000
19. ASSEMBLY CONTAINS WARNINGS: - FLAGS: U
20. END MASM - LINES: 20 TIME: 2.975 STORAGE: 28970 WARNINGS: U(2)

```

## Explanation

### Line 4

Is interpreted as a decimal number.

### Line 5

Is interpreted as an octal number because the \$OCTAL directive (default) is in effect.

### Lines 7-8

Are interpreted as hexadecimal numbers because the \$HEX directive is in effect.

### Line 11

Contains label references a0 and abc14. These symbols are not available in the MASM dictionary and must be resolved at collection time.

## 4.2.1.1. The Flag Attribute

The flag attribute of a value, also referred to as the leading asterisk flag, is clear for the results of most of the operators described in 4.3, even if one or more of the operands had the flag set. The unary asterisk (\*) is generally the only way to set the flag. Some built-in functions, however, return a node whose selectors can have the flag set. When computing with flagged values, be careful not to test for the flag on the results of such computations.

The flag attribute is generally used only on values that are selectors of node references. Therefore, the flag attribute for a selection is tested by prefixing the last selector with an asterisk (for example, ABC(1,3,\*5)). The built-in function \$IBITS (see 4.4.3) is available to test for the flag.

### 4.2.1.2. Relocation

When a program is being assembled, MASM must allocate storage to all the data items in the program and to all the instructions in the program. A location counter is used to keep track of the current relative location of data or instructions being assembled. These locations or addresses are assigned from 0, the beginning of the location counter. Such addresses are called relative addresses because they are relative to the beginning of storage. Each time a storage location is used for some instruction or piece of data, the location counter currently being used is incremented.

A reference to a name not defined in the program currently being assembled creates an external reference. Thus, there are two types of relocation: relocation by location counter and relocation by external reference.

#### 4.2.1.2.1. \$AP(*e*) — Absolute Part

The value of \$AP(*e*), where *e* is an integer expression, is the numeric part of *e* with all relocation information deleted. None of the other attributes of *e* are affected by this function.

An expression of the form \$AP(*e*)=*e* is true only if *e* has no relocation. An expression of the form:  $e_1 - \$AP(e_1) = e_2 - \$AP(e_2)$  is true only if the relocation for *e*<sub>1</sub> and *e*<sub>2</sub> is the same.

#### Example

If ABC is at offset 047 relative to the base of location counter 1, then \$AP(ABC) is 047. The value of \$AP(\$LCV) is \$LCV-\$LCB.

#### 4.2.1.2.2. \$BA(*e*) — Integer Attributes

The value of \$BA(*e*), where *e* is an integer expression, is a node reference whose elements describe the integer attributes of *e*.

If a FORM is attached to *e*, selector 0 of the node is defined and its selectors (starting at 1) are the field sizes of the FORM.

If *e* has *m* relocation items, the selectors 1, ..., *m* of the value of \$BA are defined and each of them has from 3 to 5 subselectors based on the type of relocation and the information attached.

The first of the subselectors is the leftmost bit of relocation, the second is the rightmost bit of relocation, and the third is the relocation itself. The relocation is an integer if relocation is by a location counter. It is a string (the external reference name) if relocation is by an external reference. If the relocation is negative, the third subselector is flagged. The fourth subselector has the attributes that can be attached to a location counter or external reference name. The fifth subselector is the library code name that can be attached to an external reference name.

If  $e$  has neither a form nor any relocation, the value returned by  $\$BA$  is an empty node.

Table 4-1 summarizes selectors.

**Table 4-1. Selectors Defined on the Result of  $\$BA(e)$**

Selector	Description
(0,j)	Field size in bits of the $j$ th field of the attached FORM, if any. FORM fields are counted from the left.
(i,1)	Bit number of the leftmost bit in the relocated field. Bits are numbered from right to left, starting at 0.
(i,2)	Bit number of the rightmost bit in the relocated field.
(i,3)	If relocation is by a location counter, this is the number of that location counter. If relocation is by an external reference, this is a string whose characters are the name of the external symbol.
(i,*3)	If relocation is to be subtracted, this value is one.
(i,4)	The encoded attributes that can be attached to a location counter or external reference name. (See 10.3.1 and 10.4.1).
(i,5)	The library code name that can be attached to an external reference name. (See 10.4.3).

### Example

Assume label EOR is external and the following lines are interpreted:

```

ABC      $EQU      +(J EOR)
Z        $EQU      $BA(ABC)

```

The value of Z is as follows:

```
$L0($L1(6,4,4,4,2,16),$L1(15,0,'EOR'))
```

### 4.2.1.3. Parenthetic Expression Items

An expression can be enclosed by parentheses and be preceded by an operator. Such an expression is known as a parenthetic expression. Its primary function is that of algebraic grouping.

### Example 1

`4*(5+2)`

The parenthetic expression (5+2) is used to add 5 and 2 before multiplying the intermediate result by 4.

### Example 2

`+(6+10)`

The plus sign (+) preceding the parenthetic expression (6+10) is used to prevent literal generation.

## 4.2.1.4. Literal Items

A literal is an expression enclosed in parentheses. With the exception of the unary \* and the conditional operators described in 4.3, no operators can be at the same level as the enclosing parentheses. Literals are essentially line items without the preceding operator. Since literals are relocatable values, all the rules that apply to relocatable values apply to literals. They usually have an attached FORM as well. To use the value of a literal in an expression, an extra set of parentheses is required. The expression `46+(J BEGIN)` does not cause a literal to be generated.

The value of a literal on a summary pass is zero. It is given its true value only in the generative pass. Therefore, use caution when testing the value of a literal for conditional interpretation (that is, used before the `->` operator or as an operand of a `$DO` or `$IF` directive).

### Example 1

`A        $EQU        (ADR1,ADR2)`

The symbol A is associated with the address of the literal (ADR1,ADR2).

### Example 2

`(1, TABLE)`

The address of the literal (1, TABLE) is generated.

### Example 3

`LR        R5, (PVM,15 2, ABBC)`

The address of the literal (PVM,15 2, ABBC) is used as an input parameter to the control information associated with the symbol LR.



### Example 4

```
AZ      $EQU      (F12618 1,14,ADDR)
```

The symbol AZ is associated with the address of the literal (F12618 1,14,ADDR).

### Example 5

```
Z      $EQU      +(ADR1,ADR2)
```

The symbol Z is set to the value resulting from the evaluation of expression +(ADR1,ADR2). No literal is involved.

## 4.2.1.5. Line Items

A line item is an expression involving some manipulation other than the operators listed in 4.3. The information within the parentheses must be a valid MASM line, exclusive of the label. This means that an operation field (possibly void) must be present. A line item can thus reference an instruction, a procedure, a FORM, or can generate only data. If a procedure is called, it cannot increment the current location counter; the current location counter is said to be blocked. The procedure called can call other procedures inside its own line items; in this way, a number of location counters can come to be blocked. An attempt to alter a blocked location counter results in a T flag.

To detect a line item, MASM evaluates the first expression following a left parenthesis. If the next character after the expression is not a right parenthesis, the character must be a comma or a space. If it is a comma and no significant space (not following an operator) is found before the right parenthesis, an implicit call to \$GEN is made. Otherwise, the first expression after the left parenthesis must be a directive, instruction, or procedure call. This means MASM recognizes the format (LA,U A0,1) as a valid line item.

### Example 1

```
046+(J TAG)
```

The line item, (J TAG), is an OS 1100 instruction mnemonic that, when generated, is added to the constant 046.

### Example 2

```
024+(1,2,3,6)
```

The designated word size is divided into four equal parts and each expression of the line item, (1,2,3,6), is placed into the parts of the word. The result is added to the constant 024.

### Example 3

```
054+(PVM,U TAG)
```

The line item (PVM,U TAG) calls procedure PVM. The result is then added to the constant 054.

## 4.2.2. Floating-Point Values

Floating-point values are used less frequently than integer values. MASM maintains floating-point numbers internally with a 12-bit characteristic, a 70-bit mantissa, and two sign bits, irrespective of the final precision of the number to be generated.

Floating-point numbers can be either decimal, octal, or hexadecimal; the choice between octal and hexadecimal depends on the \$OCTAL or \$HEX directive in effect. A floating-point number must contain a decimal point (.) that can be the first character, the last character, or an embedded character. If the decimal point is the last character in a floating-point number, a blank can not follow it; because in this case the decimal point is interpreted as a period and the start of a comment field rather than part of the number. The term decimal point encompasses the octal point or hexadecimal point when a floating-point number is represented in either of these bases.

A decimal floating-point number can begin with zero only if the next character is the decimal point. Octal or hexadecimal floating-point numbers always begin with a zero followed by a digit. Other rules for integer numbers apply, particularly that for a trailing D in hexadecimal mode.

### Example 1

```
1.0
```

This value is a decimal floating-point number.

### Example 2

```
0.7
```

This value is a decimal floating-point number because the 0 is followed immediately by a decimal point.

### Example 3

```
00.7  
011.7  
00.00006
```

These values are octal floating-point numbers if the `$OCTAL` directive is in effect.

### Example 4

```
00.8  
0A.4B
```

These values are hexadecimal floating-point numbers if the `$HEX` directive is in effect.

## 4.2.3. String Values

MASM treats a string as an independent data type. MASM converts a string to integer only to generate output data or when required to by the context of the expression. In addition to string constants, some of the built-in functions return string values.

To write string constants, code an apostrophe character (') followed by any combination of characters (other than an apostrophe), and terminate with another apostrophe character. To include a single apostrophe character in a string, enter two apostrophe characters (") where you want to generate an apostrophe. For a description of how to continue a string onto a following line, see 2.3.4.

### Example 1

```
A      $EQU      'A'R
```

The symbol `A` is associated with the value `06`, with string attributes indicating right justification. This example assumes the system character set is `Fieldata`.

### Example 2

```
A      $EQU      'ABCDEF'DL
```

The symbol `A` is associated with the value `060710111213`, with string attributes indicating left justification and double precision (defined in 4.3.1.5). If generated, this produces a 12-character string. This example assumes the system character set is `Fieldata`.

### Example 3

```
STG      $EQU      'abc'
```

The symbol STG is associated with the value 0141142143, with string attributes indicating right justification and single precision. If generated, this produces a 4-character string. This example assumes the system character set is ASCII.

### 4.2.3.1. String Limits

The string length limit for strings in MASM depends on the character size and how the strings are stored internally. The limits are shown in the following table:

Limit	String Type
2041 words	A string EQU'd to a 2-word label
2042 words	A string EQU'd to a 1-word label
2044 words	Generated strings

This means the string limit is from 12,251 to 12,263 Fieldata characters, depending on how the string is stored internally. The ASCII limits are 8,164 to 8,176 characters.

These limits are determined by MASM internal storage packets, which are limited to 2,047 words. From 2,041 to 2,044 words are available for the string depending on what other information is saved with the string.

### 4.2.3.2. Defining a Character Set

This subsection describes the directives that are available to define a desired character set.

#### 4.2.3.2.1. \$ASCII — Set Character Mode to ASCII

This directive requires no parameters. It sets the system character set to ASCII. This directive has special interaction with the \$CHAR directive.

#### 4.2.3.2.2. \$FDATA — Set System Character Set to Fieldata

The \$FDATA directive requires no parameters. It sets the system character set to Fieldata and has the synonym FIELDATA. This directive has special interaction with the \$CHAR directive (4.2.3.2.3).

#### 4.2.3.2.3. \$CHAR — Define a Data Character Set

The format of the \$CHAR directive is as follows:

$$\$CHAR, e_0 \quad e_i, f_i, \dots, e_n, f_n$$

where

$e_0$

indicates character frame size and is converted by the rules for parameter conversion to a positive number from 1 to 36. The character frame size for Fieldata is 6 bits and the character frame size for ASCII is 9 bits.

Each ordered pair  $(e_i, f_i)$

indicates current and future character codes, respectively. The pairs are converted by the rules for parameter conversion to nonnegative numbers.

If  $e_0$  is void, the previous character frame size remains in effect.

If all parameters are void, any data character set is inactivated and MASM reverts to the system character set.

The translation table is constructed in one of the following ways:

- If no data character set is effective before using the \$CHAR directive, a table is constructed to translate each character of the system character set into the character of the new data character set.
- If there is a data character set with a translation table constructed for the same system character set, a copy of that table is used.
- If there is a data character set with a translation table constructed for a system character set different from the one specified by the current \$CHAR directive, the translation table is constructed in two steps. Each character of the current system character set is translated as follows:
  - Translated into the equivalent character of the previous system character set
  - Translated through the translation table into the final code

Once the table is constructed, the parameter pairs  $e_i, f_i$  are taken in order and modifications are made to the table to indicate that the character of the system character set with the code  $e_i$  translates into  $f_i$ .

If the system character set is Fieldata, the number of pairs cannot exceed 64. If the system character set is ASCII, the number of pairs cannot exceed 128. No value of  $f_i$  can exceed  $1*/36-1$ .

The translate table need not be completely redefined to alter only a few characters of it.

The existence of a character translation table overrides the setting of the system character set by the \$ASCII or \$FDATA directives.

### Example 1

```
1.      $CHAR 'A',077
2.      + 'AB'
3.      $ASCII
4.      + 'AB'
5.      $CHAR
6.      + 'AB'
```

### Explanation 1

This example assumes the system character set is Fieldata.

#### Line 1

Builds a character translation table that converts a value of 06 to 077.

#### Line 2

Shows MASM code to generate 07707, using the character translation table.

#### Line 3

Shows the \$ASCII directive setting the system character set to ASCII.

#### Line 4

Generates a value of 07707 because the character translation table overrides the system character set. The character translation table consists of the Fieldata character set with one modification: 06 has been converted to 077.

#### Line 5

Shows a \$CHAR directive with no parameters removing the character translation table.

#### Line 6

Shows MASM generating a value of 101102 because the system character set is ASCII.

### Example 2

```
1.      $ASCII
2.      $CHAR 'A',071
3.      + $CFS('ab')
4.      + 'ab'
5.      $FDATA
6.      $CHAR 'A',06
7.      + 'AB'
8.      + '99'
```

### Explanation 2

Line 1

Shows the \$ASCII directive setting the system character set to ASCII.

Line 2

Builds a character translation table that converts a value of 101 to 071.

Line 3

Generates a value of 071102, using the character translation table. The \$CFS function converts the ASCII string 'ab' to the Fielddata string 'AB', which is then converted via the character translation table.

Line 4

Generates a value of 0141142, using the character translation table.

Line 5

Shows the \$FDATA directive setting the system character set to Fielddata.

Line 6

Shows a \$CHAR directive creating a new character translation table as follows:

- Each character of the Fielddata character set is translated into its ASCII equivalent and then into the final code through the previous translation table. The new character translation table consists of the Fielddata character set translated to ASCII with one modification: 101 has been converted to 071.
- The parameter pair 'A',06 is evaluated and the translation table is modified to convert a value of 071 to 06. The Fielddata character with the code 071 is translated to 06, not the character 'A', which has a Fielddata code of 06.

Line 7

Generates a value of 071102, using the character translation table.

Line 8

Generates a value of 006006, using the character translation table.

### 4.2.3.3. String Conversion Functions

The functions described in this subsection create strings from other data types.

#### 4.2.3.3.1. \$CAS(e) — Convert to ASCII String

The expression *e* is converted to the ASCII character set. If *e* is a string, the string is converted from its original character set to its ASCII equivalent. If the expression *e* is an integer value, it is formed into 9-bit groups with the lower eight bits as significant bits; the resultant value is marked as an ASCII string.

### Example 1

```
+      $CAS('AbCd')
```

If the system character set is Fielddata, then the string AbCd is converted to ASCII and the output is as follows:

```
0101102103104
```

If the system character set is ASCII, the string AbCd is “converted” to ASCII and the output is as follows:

```
0101142103144
```

### Example 2

```
+      $CAS(012040777)
```

The integer value 012040777 is formed into 9-bit groups, with the lower eight bits as significant bits. The resulting value is as follows:

```
012040377
```

This example produces a T flag because bits are lost when only the eight least significant bits are output.

#### 4.2.3.3.2. \$CFS(*e*) — Convert to Fielddata String

The \$CFS function converts the expression *e* from its original character set to the current Fielddata character set. If the expression *e* is an integer value, the value is formed into 6-bit groups, and the resultant value is marked as Fielddata.

#### 4.2.3.3.3. \$CS(*e*) — Convert to String

The \$CS function converts the expression *e* from its original character set to the data character set, if any. If there is no data character set, the result is in the system character set.

#### 4.2.3.3.4. \$CD(*e*) — Convert to Decimal

If *e* is an integer value without relocation, the value of \$CD(*e*) is a string in the system character set that contains the decimal representation of *e*. If *e* is negative, a leading minus (-) is present. The length of the result is the minimum necessary to contain the significant digits of *e*.



## Example

```

1. @MASM,SR MASM*MSM4.VA002,TPF$.
2. MASM xxRyy
3. 0 000000 616060050505
4. 000001 000000616060
5. U 0000000000014
6. ** BITS 72-0 + XR XREF
7. R 000002 000000006162
8. 000003 000000006162
9.
10.
11. LOCATION COUNTERS: $(0) 000004 $(1) 000000
12. ASSEMBLY CONTAINS ERRORS: - FLAGS: RU
13. END MASM - LINES: 12 TIME: 2.925 STORAGE: 28970 ERRORS: R(1) /U(1)

```

## Explanation

### Line 3

The value of \$CD(0144) is the string '100' or 616060050505.

### Line 4

The value of + \$CD(0144) is the string + '100' or 000000616060.

### Line 5

The label TAG is equated to 12 plus the value of the undefined reference XREF. The second line, showing the relocation attached to TAG, is listed using the R option on the MASM call.

### Line 7

The value of \$CD(TAG) produces an R flag if TAG has relocation attached.

### Line 8

The value of \$CD(\$AP(TAG)) is the absolute value of TAG converted to a string. No R flag is issued because the \$AP function removes the relocation attached to TAG before the \$CD function converts the resulting parameter.

## 4.2.3.3.5. \$CB — Convert to Integer Representation

The \$CB( $e_1, e_2$ ) function requires  $e_1$  and  $e_2$  to be integers, with  $0 \leq e_2 \leq 25$ .  $e_2$  is the string length. If  $e_2$  is omitted, a value of zero is assumed. The result of the \$CB function is a string in the system character set that represents the value of  $e_1$  in the integer representation mode.

The significant digits are left-justified within the string if spaces must be added to fill the word. Leading zeros are added to the string so the length of the resulting string equals the requested length  $e_2$ . At least one leading zero is in the resulting string. If the parameter  $e_2$  is less than the number of characters resulting from the conversion of parameter  $e_1$  (plus the leading zero), the length of the string is the minimum necessary to convert the parameter.

### Example

```
1. @MASM,S MASM*MSM4.VA003,TPF$.
2. MASM xxRyy
3.      0 000000 606164640505      1.          $cb(0144,1)
4.      000001 606060616464      2.          $cb(0144,6)
5.      000002 416060606164      3.          $cb(-0144,7)
6.      000003 640505050505
7.
8.
9. LOCATION COUNTERS:      $(0) 000004      $(1) 000000
10. END MASM - LINES: 8    TIME: 2.909    STORAGE: 28970
```

### Explanation

#### Line 3

The resulting string is '0144' padded on the right with blanks in order to fill the generated word size.

#### Line 4

The resulting string is '000144'.

#### Lines 5-6

The resulting string is '-000144' plus 5 blanks to fill out the two words generated.

## 4.2.3.4. String Manipulation Functions

This subsection describes the character string functions that are available.

### 4.2.3.4.1. \$SL(*e*) — String Length

The parameter *e* must be a string. The value returned is the number of characters in the string *e*.

#### Example

```
$SL('SIX+ONE')
```

The value returned by the function is 7.

### 4.2.3.4.2. \$SR(*e1*,*e2*) — String Repetition

The \$SR function returns a string that is constructed by concatenating *e2* copies of the string *e1*. Thus, *e2* should be a nonnegative integer value without relocation. If *e2* is zero, a void string is returned.

#### Example

```
$SR('ABC',3)
```

The value returned by the function is 'ABCABCABC'.

### 4.2.3.4.3. **\$SS(e1,e2,e3) — Substring Extraction**

For the \$SS function,  $e_1$  is a string and  $e_2$  and  $e_3$  are integers with  $e_2 \geq 1$  and  $e_3 \geq 0$ . If  $e_3$  is omitted, 1 is used. \$SS returns the substring of  $e_1$  starting at character  $e_2$  (numbered from the left beginning with 1) for a length of  $e_3$ . If  $e_3$  requests more characters than are present to the end of string  $e_1$ , the result is blank filled to  $e_3$  characters. If  $e_3$  is zero, the result is a void string.

This function can be used to left-justify strings within a given field size.

#### Example 1

```
$SS('Example',3,3)
```

The result is the string 'amp'. This example assumes ASCII.

#### Example 2

```
$SS('A',1,10):$SS('EQU',1,10):'1'
```

The result is the string, 'A EQU 1'. A begins in position 1, EQU in position 11, and 1 in position 21.

### 4.2.3.4.4. **\$SSS(e1,e2,e3,e4) — Substring Substitution**

For the \$SSS function,  $e_1$  and  $e_2$  are strings, while  $e_3$  and  $e_4$  are integers, with  $e_3 \geq 1$  and  $e_4 \geq 0$ . If  $e_4$  is omitted, a value of 1 is assumed. The value of \$SSS is a string constructed by substituting the string  $e_2$  for the  $e_4$  characters of string  $e_1$  beginning at character  $e_3$  of  $e_1$ . The other portions of the result are the rest of string  $e_1$ . The string  $e_1$  is extended by blanks on the right, if necessary, to make the expression meaningful. If  $e_4$  is zero, the insertion is done before character  $e_3$ . If  $e_2$  is void, this function deletes a substring of  $e_1$ . The type of the resulting string is determined by the following rules in order of precedence:

1. If  $e_1$  or  $e_2$  is data character set, the result is the data character set.
2. If  $e_1$  or  $e_2$  is ASCII, the result is ASCII.
3. If  $e_1$  or  $e_2$  is Fielddata, the result is Fielddata.

The result of the \$SSS function can also be computed using the \$SS function and concatenation operators; however, \$SSS is clearer. This function, when combined with the \$DISPLAY directive, can construct commentary display at assembly time.

#### Example

```
$SSS('ABCDEF','HIJ',3,2)
```

The value returned by the function is 'ABHJEF'.

### 4.2.4. Nodes and Selectors

A node is a point of departure for a tree structure. Each node can have a set of selectors defined for it. The number of selectors is limited only by the amount of available storage. Each selector is defined by a unique nonnegative integer. The integers used need not be in any particular sequence. To obtain a particular selector of a node, write the number of the selector, in parentheses, following the node reference. If the value of the selector is itself a node, make further selections by writing the number, separated by commas, of each selection within the same set of parentheses. The selector number can be computed by MASM expressions.

A selector can be either a value or a node reference. The node reference can also have a set of selectors defined for it.

The value given to any particular selector of a node can be any legitimate MASM value, including another node. The values of various selectors of the same node need not be of the same type. If a node reference is used in a context requiring a numerical value, the value of the node reference is the number of selectors defined. As with other MASM values, node references can be passed as arguments and assigned to labels. A label whose value is a node reference can be referenced as a subscripted label (see 2.3.3.1). If a node reference is passed as a parameter to a procedure, the number of subscripts of the associated structure is 2 plus the number permitted for any selection sequence of the node reference itself.

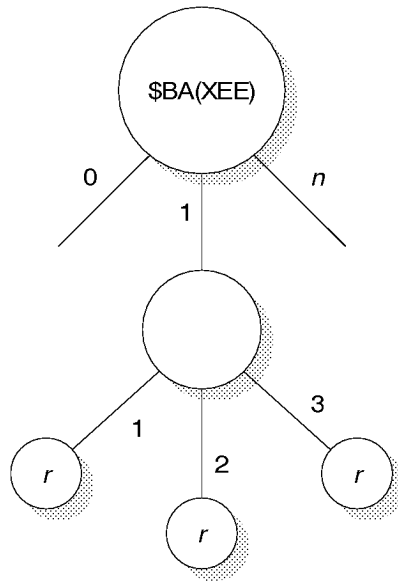
Node references are created by the procedure parameter passing mechanism, some built-in functions, and the simple writing of a subscripted label in the label field, where the label was not previously defined. Node references can be deleted by removing all references to them through reassignment or by using the `$DELETE` directive. Since the ordinary equality test of MASM tests only for numerical equality and hence is satisfied by two nodes with the same number of selectors, a pair of operators is provided that tests two nodes for exact identity (and nonidentity). That is, the test is satisfied only if the operands are the same node.

#### 4.2.4.1. Examples of Node References

##### Example 1

$\$BA(XEE)(1)$

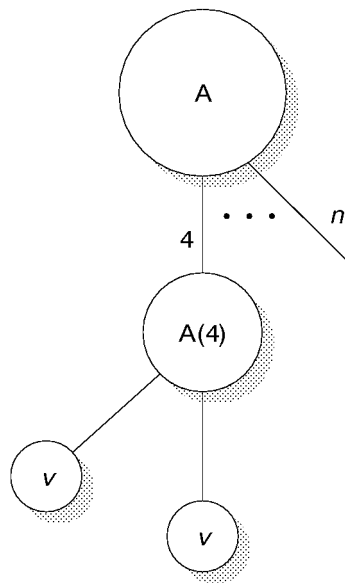
Selector 1 of the node  $\$BA(XEE)$  points to a node; therefore, example 1 is a node reference. (See 4.2.1.2.2.)  $r$  indicates relocation information.



### Example 2

$A(4)$

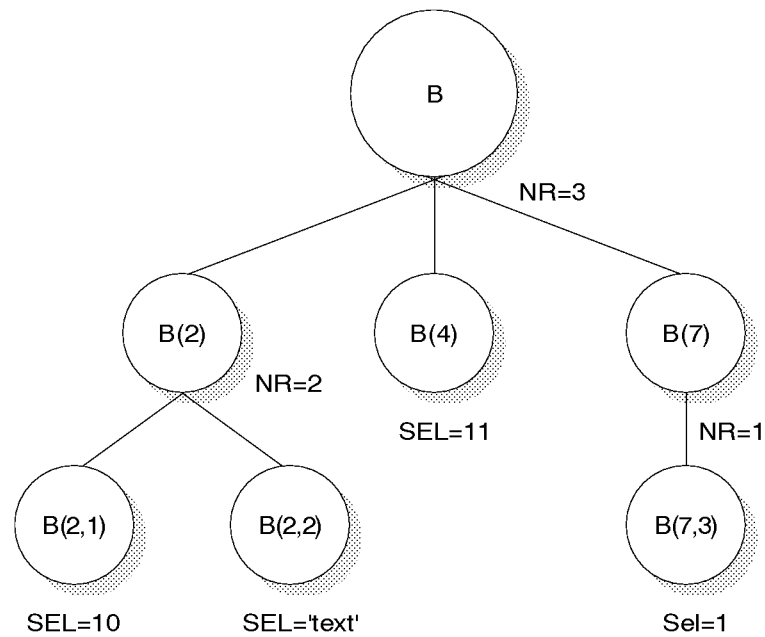
Assume  $A$  is a node with at least one selector, 4, which also has selectors. Then  $A(4)$  is a node reference and  $v$  indicates some value.



## Example 3

1.		1.	b	\$equ	\$node
2.	000000000012	2.	b(2,1)	\$equ	10
3.		3.	b(2,2)	\$equ	'text'
4.	000000000013	4.	b(4)	\$equ	11
5.	000000000001	5.	b(7,3)	\$equ	1

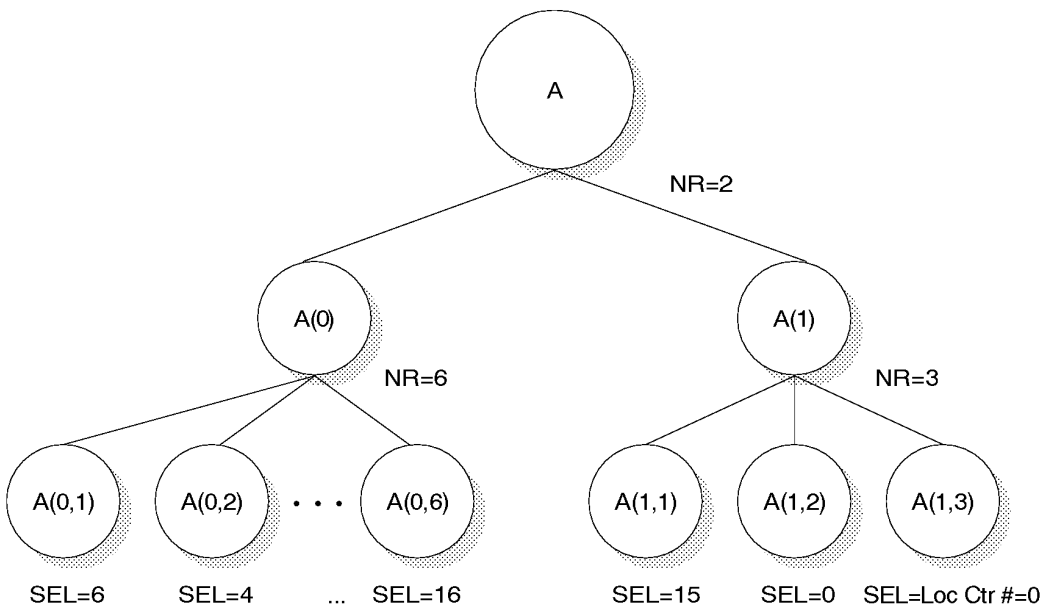
B is a node with assigned node references and selector values. NR is the number of assigned node references and SEL is a selector value.



Example 4

1.		1.	axr\$
2.		2.	tag
3.	10 00 00 00 0 000000	3.	abc      \$equ      +(1a a0,tag)
4.		4.	A        \$equ      \$ba(abc)

Tag A is equated to a node structure of assigned node references and selector values generated by the \$BA function (see Table 4-1). NR is the number of assigned node references and SEL is a selector value.



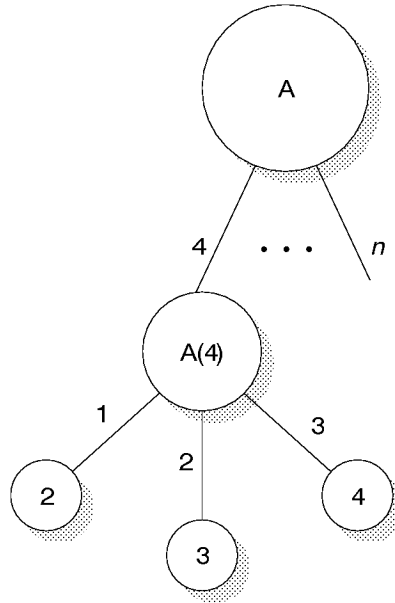


#### 4.2.4.2. Examples of Node Reference Selectors

##### Example 1

$A(4,1)$

A graphic representation of this follows:

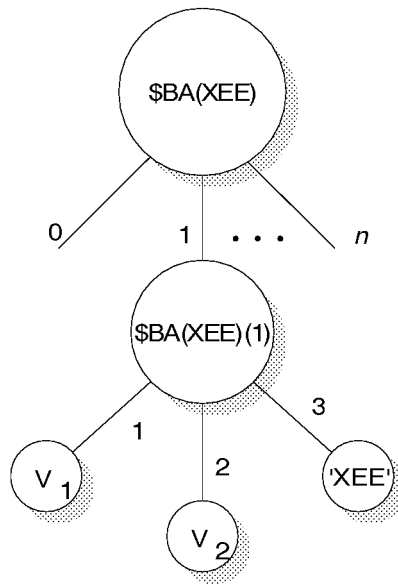


If the node  $A(4)$  had three values associated with it and they were 2,3,4, then the value of selector  $A(4,1)$  would be 2.

### Example 2

$\$BA(XEE)(1,3)$

A graphic representation follows:



If XEE is undefined, the value associated with  $\$BA(XEE)(1,3)$  is the string 'XEE' (see 4.2.1.2.2).

## Example 3

```

1. @MASM,S MASM*MSM4.N0001,TPF$.
2. MASM xxRyy
3.
4.      000000000024
5.      000000000012
6.      000000000005
7.      000000000006
8.
9.
10. 0 000000 000000000003
11. 000001 000000000001
12. 000002 000000000024
13. 000003 000000000000
14. 000004 000000000001
15. 000005 000000000001
16. 000006 000000000002
17. 000007 000000000005
18. 000010 000000000006
19.
20.
21. LOCATION COUNTERS: $(0) 000011 $(1) 000000
22. END MASM - LINES: 34 TIME: 3.448 STORAGE: 28918

```

1. A	\$EQU	\$NODE
2. A(1)	\$EQU	20
3. A(3,2)	\$EQU	10
4. B(2,1)	\$EQU	5
5. B(2,3)	\$EQU	6
6. A(4,1)	\$EQU	B
7. .		
8. +		A
9. +		B
10. +		A(1)
11. +		A(2)
12. +		A(4)
13. +		A(4,1)
14. +		A(4,1,2)
15. +		A(4,1,2,1)
16. +		A(4,1,2,3)
17. \$end		

## Explanation

### Line 3

A is explicitly defined as a node reference.

### Line 4

Selector (1) of the node reference A is equated to the value 20.

### Line 5

Selector (3,2) of node A is equated to the value 10.

### Line 6

Selector (2,1) of node B is equated to the value 5. Node B is implicitly defined as a node reference because of the subscripted label B.

### Line 8

Selector (4,1) of node A is equated to the node reference B.

### Line 10

Three selectors are defined for node A (1, 3, and 4).

### Line 11

One selector is defined for node B (2).

### Line 12

The selector A(1) is equated to the value 024.

### Line 13

The selector A(2) does not exist.

Line 14

The selector A(4) has one selector defined (1).

Line 15

The selector A(4,1) has one selector defined (2).

Line 16

The selector A(4,1,2) has two selectors defined (1, and 3).

Line 17

The selector A(4,1,2,1) is equated to the value 5, which is the same as B(2,1).

Line 18

The selector A(4,1,2,3) is equated to the value 6, which is the same as B(2,3).

### 4.2.4.3. \$L0(e1,...,en) — Form a List Starting at 0

The parameters  $e_1, \dots, e_n$  are converted according to the rules for parameter conversion. A new node is constructed whose  $n$  selectors run from 0 to  $n-1$  and whose  $i^{\text{th}}$  selector value is  $e_i$ . This function can be used to construct complex tree structures such as those found in LISP, SNOBOL, or PL/I.

#### Example

```
A      $EQU      $L0('X',6,$L0(014))
```

A is defined as a node reference with three selectors defined (0,1,2). Selector A(0) is equated to the character 'X'. Selector A(1) is equated to the value 6. Selector A(2) is defined as a node reference with one defined selector (0). Selector A(2,0) is equated to the value 014. No other selectors of A are defined.

### 4.2.4.4. \$L1(e1,...,en) — Form a List Starting at 1

The \$L1 function performs an operation identical to that of \$L0, except that the first defined selector of the result is 1 rather than 0.

### 4.2.4.5. \$NODE — Form a Node

The \$NODE function requires no parameters. It returns a reference to a node that is distinct from every other node so far created. The node thus produced has no selectors defined. This can be required if the identifier is assigned a value other than a node and its content expected a node value.

For example, the following lines are not accepted when interpreted by MASM:

```
A      $EQU      15
Y      $EQU      A(3)+1
```

The lines are not accepted because using A with a subscript is illegal, as in A(3). The following lines will achieve the desired results.

```
A      $EQU      $NODE
Y      $EQU      A(3)+1
```

#### 4.2.4.6. \$NS(e1,e2) — Find nth Selector

A nonempty node can have any set of selector numbers in use, which do not need to be successive integers or start at any particular integer. The \$NS function requires that  $e_1$  is a node reference and  $e_2$  is an ordinal integer less than or equal to the number of selectors defined for  $e_1$ . The value of  $\$NS(e_1, e_2)$  is the value of the  $e_2^{\text{th}}$  selector of  $e_1$ . The selectors of  $e_1$  are ordered sequentially so that  $e_2$  begins with 1. An error results if  $e_2$  is larger than the number of selectors defined for  $e_1$ . (The first selector is found when  $e_2 = 1$ .)

Since the selectors for a node are ordered by increasing value, the selection mechanism can be used for sorting. For example, if each sort key K has an associated value VK, then you can initialize

```
A      $EQU      $NODE
```

and then perform for each key K:

```
A(K)  $EQU      VK
```

The sorted values can be retrieved in order by referencing the selector numbers  $\$NS(A,1)$ ,  $\$NS(A,2)$ , and so on.

#### Example

If A(2) and A(4) are the only selectors defined for A, then  $A(\$NS(A,1))$  is the same as A(2), and  $A(\$NS(A,2))$  is the same as A(4).

#### 4.2.4.7. \$SN(e1,e2) — Find Selector Number

The \$SN function is the converse of the \$NS function. Where \$NS uses an ordinal number to return the defined selector for a node, \$SN uses a defined selector  $e_2$  for the node  $e_1$  and returns its appropriate ordinal. Therefore,  $e_1$  must be a node reference and  $e_2$  should be an integer value without relocation. If the selector  $e_2$  is defined for  $e_1$ , the value of  $\$SN(e_1, e_2)$  is the ordinal number of that selector. Otherwise, the value is 0.

The expression  $\$SN(e_1, e_2)$  is 1 if  $e_2$  is a selector for  $e_1$ ; otherwise, the value is 0.

### Example

If A(2) and A(4) are the only selectors defined for A, then \$SN(A,2) is 1 and \$SN(A,4) is 2.

## 4.2.5. Control Information

Control information is the name used to refer to values associated with MASM directives, built-in functions, procedure names, and function names. This means values that are not associated with data, but control some stage of processing. Control information requires a special context, usually one that is associated with the operation field of a MASM line or line item.

Any MASM symbol (or node reference selector) can have both a normal data value and a control information value. The value used by MASM at any given time depends on the context being processed. Thus, the operation field of a MASM line (or line item) retrieves the control information associated with an expression, while other contexts require a data type value. The \$EQU directive assigns both values; thus, this directive can be seen to have a special context. The slash (/) forces the context for the following expression to be for control information (the expression is usually a label or other elementary item). This operator can be used to pass a directive into a procedure.

### Example 1

```
ADD      $EQU      AA
```

The symbol ADD can be used as the Series 1100 instruction mnemonic AA (Add to A).

### Example 2

```
MACRO    $EQU      $PROC
```

The symbol MACRO can be used as the MASM directive \$PROC.

### Example 3

```
PVM      14, /LA
```

Assuming the symbol PVM is a procedure call, then the symbol LA is passed into the procedure. Without the prefix operator (/), the symbol LA would be evaluated with the procedure receiving the resultant value.

## 4.3. Expressions and Operators

A powerful set of operators manipulates assembly-time values. Values and operators make up expressions. Evaluating expressions produces new values. MASM contains both unary and integer operators. Some of these operators use the same symbol. Where a symbol has more than one meaning, the context determines which operator is intended.

The syntax for MASM expressions is similar to most high-level languages. The integer operators all use infix notation (operand/operator/operand). Some of the unary operators use prefix notation (operator/operand), and others use postfix (operand/operator). There is also a conditional operator with a more complex syntax described in 4.3.1.5.2.

The hierarchy of operators in MASM follows the usual rules for arithmetic and logical operators. See 4.3.2.1 for a complete description of operator precedence in MASM.

Because all MASM arithmetic is performed in high precision, the programmer does not need to be concerned with single-precision and double-precision mixed-mode arithmetic. The distinction in precision is made only when data is generated and controlled by the programmer using the S (single-precision) and D (double-precision) operators (see 4.3.1.5.1).

### 4.3.1. Operators

The following MASM operator types are described in this subsection:

- Arithmetic
- Logical
- String
- Relational
- Other MASM operators

#### 4.3.1.1. Arithmetic Operators

MASM provides operators for arithmetic phrases and integer (binary) and floating-point numbers. MASM supports the usual arithmetic operations of sum (+), difference (-), product (\*), quotient (/), unary positive (+), and one's complement negative (-). Also, MASM provides other useful operations that are less common.

For example, the arithmetic division covered quotient operator (//) computes the arithmetic covered quotient of its two operands, and is meaningful only for integer arithmetic. The covered quotient equals the ordinary quotient if the division remainder is zero. However, if the division remainder is not zero, the covered quotient is 1 greater than the ordinary quotient.

### Example

```
5//3=2  
(-5)//3=-2
```

The arithmetic division remainder operator (`//`) computes the remainder from the arithmetic division of its operands and is meaningful only for integer arithmetic.

The fixed-point integer scaling operator (`*/`) multiplies its left-hand operand by the power of two, given by the right-hand operand. If the right-hand operand is positive, this is a left shift. If the right-hand operand is negative, this is a right (arithmetic) shift. An error results when either operand is floating-point (T flag), the second operand is relocatable (R flag), or the generated value is too large (T flag).

The floating-point power of 10 positive scaling operator (`*+`) converts its left-hand operand to floating-point, if necessary, and multiplies the operand by the power of 10 given by the right-hand operand. The right-hand operand cannot be floating-point and neither operand can be relocatable. The result is always floating-point.

The floating-point power of 10 negative scaling operator (`*-`) converts its left-hand operand to floating-point, if necessary, and divides it by the power of 10 given by the right-hand operand. The right-hand operand cannot be floating-point and neither operand can be relocatable. The result is always floating-point.

The floating-point power of 2 scaling operator (`*/`) performs floating-point scaling by a power of 2. It is used primarily with octal or hexadecimal floating-point numbers. This operator is similar to the (`*+`) operator except that multiplication is by a power of 2 (possibly negative) instead of a power of 10.

MASM issues error flags when illegal operations are requested. Generation of values greater than 72 bits or division by zero generates truncation errors (T flag). Illegal operation on relocation (see 4.2.1.2) generates relocation errors (R flag) and the relocation information is lost. Inappropriate values (such as procedures and forms) used in arithmetic expressions generate value errors (V flags). V flags are issued only if the value cannot be converted to a number. (See 4.3.2.2 for a discussion of conversion rules.)

### 4.3.1.2. Logical Operators

MASM provides four operators to perform bit-by-bit logical operations on integer (binary) values with a maximum of 72 significant bits. Any relocation is discarded and an R flag is issued. The result is single precision only if both operands are single precision. When only one of the operands has a form (see 5.4) or both of the operands have the same form, the result of the operation retains that form. Otherwise, the result has no form.

MASM supports three integer operations: OR (`++`), exclusive OR (`--`), and AND (`**`). The truth tables in Table 4-2 apply bit-by-bit operations to the operands to produce the result.



The NOT operator (\) produces its results differently. It is not a bit-by-bit negation. Rather, if the operand is nonzero, the result is 0; and if the operand is 0, the result is 1. The unary arithmetic negation operator (-) is different from the NOT operator (\) in that it returns a one's complement result (that is, bit-by-bit negation). Two consecutive NOT operators convert any nonzero value to 1 but leave 0 intact.

**Table 4-2. Truth Tables for Logical Operations**

Values of A	Values of B	A++B	A--B	A**B	\B
0	0	0	0	0	1
0	1	1	1	0	0
1	0	1	1	0	
1	1	1	0	1	

## Example

```

1. @MASM,S MASM*MSM4.OP001,TPF$.
2. MASM xxRyy
3.          000000000007
4.          000000000006
5.          000000000001
6.          000000000000
7.          000000000001
8.          777777777772
9. 0 000000 777777777777
10.
11.
12. LOCATION COUNTERS: $(0) 000001 $(1) 000000
13. END MASM - LINES: 14 TIME: 4.107 STORAGE: 8014

1.          $DISPLAY 5++3
2.          $DISPLAY 5--3
3.          $DISPLAY 5**3
4.          $DISPLAY \5,\0
5.          $DISPLAY -5
6.          - 0
7.          $END

```

## Explanation

Line 3

Displays the number 7

Line 4

Displays the number 6

Line 5

Displays the number 1

Line 6

Displays the numbers 0 and 1

Line 8

Displays the number 0777777777772

Line 9

Generates the number 077777777777

### 4.3.1.3. String Operators

The infix operator, string concatenation (:), is used to concatenate strings. String concatenation has following general form:

$$e_1 : e_2 : e_3 : \dots : e_n$$

where  $e_1, \dots, e_n$  are expressions with higher operator precedence (see 4.3.2.1) that convert to strings. The value of the expression is the result of concatenating the strings  $e_1, \dots, e_n$ . If any of the strings is in a data character set, then all of the strings are converted to a data character set. If there is currently no data character set, or if the character sizes of the resulting strings fail to match, an error is noted. Otherwise, if any of the strings is in ASCII, then all the strings are converted to ASCII. After the conversions, the strings are concatenated in the order in which they appear. The justification and space attributes for the resulting string are those of the last operand  $e_n$ . If any of the operands is double precision, then the result is double precision; otherwise, the result is single precision.

The concatenation operations in an expression are performed all at once (other operators and parentheses permitting) to save the storage otherwise required for intermediate results.

The L postfix operator converts its operand to a string and sets the justification attribute to left-justify, space-fill.

The R postfix operator converts its operand to a string and sets the justification attribute to right-justify, zero-fill. The justification attribute is effective when the value is generated in the output file.

#### Example 1

```
'const'R
```

This operand describes a string of length five with the right-justification flag set.

#### Example 2

```
FCTN('X'L)
```

Assuming FCTN is a function, the string 'X' (with left-justification flag set) is passed as the argument.

#### 4.3.1.4. Relational Operators

The relational operators are =, <>, <, <=, >, >=, ==, and !=. They return values of 1 or 0 when the relation specified is true or false. A simple relation has the form  $e_1 r e_2$ , where  $r$  is a relational operator and  $e_1$  and  $e_2$  are level 3 expressions. A compound relation has the following form:

$$e_1 r_1 e_2 r_2 \dots r_n e_{n+1}$$

with notation as before. The value of this expression is 1 if all of the relations  $e_m r_m e_{m+1}$  for  $m=1, \dots, n$  are true. Otherwise, the expression's value is 0. The expression  $e_1, \dots, e_n$  is evaluated only up to the first pair for which the relation is false. For this reason, some errors cannot be detected in the unevaluated portion of the level 2 expression. This does not apply to microstring substitution, which takes place before expression evaluation begins (See 6.8).

Since the modes of  $e_{m-1}$  and  $e_{m+1}$  can differ from each other and from  $e_m$ , the value of  $e_m$  can be converted twice. Both of these conversions are made from the same original value, rather than one being converted from the other. This achieves maximum consistency.

Except for the node identity (==) and node nonidentity (!=) operators that operate on node references, the relational operators operate on the integer, floating-point, and string values. For a simple relation formed from one of these operators, if one of the operands is floating-point, both operands are converted to floating-point. If neither is floating-point but one operand is an integer, then both operands are converted to integers. Otherwise, both operands are strings.

Integer and floating-point values are compared using the natural orderings of these number systems. In addition, the equal and not equal operators (= and <>) also compare relocation for integer operands. Two relocatable integer operands are equal only if the absolute part (offset) and all relocations are the same. For the operators less than, less than equal to, greater than, and greater than equal to (<, <=, >, and >=), the relocation of two integer operands must be the same or an R flag results.

When strings are compared, they are first converted to strings with the same character size. If either string is in a data character set, then both strings are converted to the data character set. If there is currently no data character set, or if the character sizes for the resulting strings differ, then a V flag is indicated. Otherwise, if one of the strings is ASCII, then the other is converted to ASCII. If either string has the left-justification attribute, then the strings are compared as left-justified strings padded to equal length with space characters. Otherwise, the strings are compared as right-justified strings padded to equal length with zeros. Once the strings are justified, they are compared according to lexical ordering based on the collating sequence formed by the character codes.

The relational equal operator (=) is true if the operands are equal and is false if they are not equal. For integer operands, the relation is true if the values are equal and the relocation matches. Otherwise, the relation is false.

The relational not equal operator (<>) is true only if the relation equal (=) is false.

The relational less than operator (<) is true if the first operand is less than and not equal to the second operand. For integer operands, the relocation of the two operands must match.

The relational less than equal to operator (<=) is true if the first operand is less than or equal to the second operand. For integer operands, the relocation of the two operands must match.

The relational greater than operator (>) is true if the first operand is greater than and not equal to the second operand. For integer operands, the relocation of the two operands must match.

The relational greater than equal to operator (>=) is true if the first operand is greater than or equal to the second operand. For integer operands, the relocation of the two operands must match.

The relational node identity operator (==) is computed by converting both operands to node references. If both node references refer to the same node, the relation is true; otherwise, it is false.

The relational node nonidentity operator (≠) is computed by converting both operands to node references. If both node references refer to the same node, the relation is false; otherwise, it is true.

### 4.3.1.5. Other Operators

This subsection describes some other types of MASM operators.

#### 4.3.1.5.1. Single- and Double-Precision Control

Two postfix operators specify the precision for generated data. Internal values are maintained with up to 72 significant bits plus a sign. Values also have a precision attribute that can be set to single or double. When the value is single precision, it is generated as one word. (The default word size of 36 bits can be changed by the \$WRD directive. See 5.7.) When the value is marked as double precision, it is generated as two words regardless of the magnitude of the number (for example, double-precision 0 is allowed).

The S postfix operator converts its operand to a value and sets the precision attribute to single precision.

The D postfix operator converts its operand to a value and sets the precision attribute to double precision. If the integer representation mode is hexadecimal, a trailing D on a hexadecimal number is interpreted as part of the number rather than as the double precision postfix operand. Therefore, use a pair of the parentheses; that is, (05FF)D.

## Example 1

```
'abc'D
```

This operand describes a string of length three with the double-precision flag set.

## Example 2

```
074D
```

This operand describes a double-precision integer when the current integer representation mode is octal.

### 4.3.1.5.2. Conditional Expressions

Conditional expressions allow the alternative generation of values without multiplying various expressions by zero or 1. The unused expression in a conditional expression is not evaluated. This permits the use of functions with side effects in cases where the zero-one multiplication technique would forbid them. Since microstrings (see 6.8) are evaluated before the evaluation of the line, this does not apply to them. On the other hand, this means that some errors cannot be detected, because the expression containing them is not evaluated.

Conditional expressions have one of the following forms:

- $a \rightarrow b!c$
- $a \rightarrow b$

The first example is complete and the second is incomplete (because the  $!c$  is missing). Conditional expressions can be substituted for  $b$  and  $c$  in the above expressions, with the restriction that the  $b$  in  $a \rightarrow b!c$  can be replaced only by a complete conditional expression. Complex conditional expressions are built this way. If all of the conditional expressions in such a construction are complete, the resulting expression is complete; otherwise, it is incomplete. With these rules, the operators  $\rightarrow$  and  $!$  in a conditional expression can be matched. As the expression is examined from left to right, each conditional else ( $!$ ) matches the most recent unmatched conditional if-then ( $\rightarrow$ ) operator. For example, the following expression

```
a->b->c!d->e!f!g->h
```

can be parenthesized as

```
a->(b->c!(d->e!f))!(g->h).
```

This means that the matching process is performed only at the same parenthesis level; parentheses create a new expression treated as a unit when MASM scans conditional expressions. The two unary operators ( $*$ ) and ( $/$ ) can appear at the beginning of a conditional expression or following a conditional if-then ( $\rightarrow$ ) or conditional else ( $!$ ) operator.

These operators (->) and (!) are used together to form conditional expressions of the form  $a \rightarrow b!c$  and  $a \rightarrow b$ . To evaluate these expressions,  $a$  is first evaluated to an integer value without relocation. If  $a$  is not equal to zero, then the value of the conditional expression is the value of  $b$ . If  $a$  is equal to zero, then the value of the conditional expression is the value of  $c$ , or void if  $c$  is not present.

Because the value of zero in MASM can be considered false and any nonzero value as true, the expression  $a \rightarrow b!c$  can be interpreted as IF  $a$  THEN  $b$  ELSE  $c$ , while  $a \rightarrow b$  can be understood as IF  $a$  THEN  $b$  ELSE *void*.

**Note:** *void is not the same as a value of zero or the null string. There are contexts where this distinction is important, such as the parameters on a \$PROC directive line.*

A conditional expression can be used to compute control information for use as a function or as a directive. Thus, the following line

```
T->LA!LNA      A0,TAG
```

generates one of the following instructions, depending on the value of T:

```
LA      A0,TAG      . (T is true)
LNA     A0,TAG      . (T is false)
```

The unary operators (\*) and (/) can appear at the beginning of a conditional expression or after any occurrence of the conditional if-then (->) or conditional else (!) operators. If these operators appear in front of a conditional expression, they are applied to the expression; otherwise, the operators are applied to the level 1 expression that they precede.

### Example 1

```
*a->b->*c!d
```

The first unary operator is applied to the whole expression; the second unary operator is applied only to the variable  $c$ .

### Example 2

```
/a->b->/c!d
```

The first unary operator is applied to the whole expression; the second unary operator is applied only to the variable  $c$ .

### 4.3.1.5.3. The Flag Attribute Operator

The unary prefix operator (\*) sets the flag attribute. This flag can be tested by the appropriate node selector reference or the \$IBITS function (see 4.4.3). The unary (\*) does not suppress literal creation.

The flag attribute is used to maintain a two-state condition. For example, the MASM built-in directive \$DISPLAY issues an E flag if its operand is flagged (see 3.3). The built-in function \$BA indicates relocation by flagging a selector in the structures it returns (see 4.2.1.2.2). You can also use the flag attribute in your own code to show any arbitrary condition.

### 4.3.1.5.4. The Control Information Operator

The unary operator (/) converts the operand to control information, if possible. If this is not possible, an error is noted, and the value of the expression is zero (see 4.2.5). The operator allows the user to pass directives as parameters without evaluating them when they pass. The control information is passed so the directive can be evaluated later.

## 4.3.2. Expressions

In MASM, operators and values (operands) comprise expressions. This subsection describes operator precedence and operand conversion rules.

### 4.3.2.1. Operator Precedence

Table 4-3 shows the hierarchy of MASM operators. Generally, these operators function from left to right in an expression. However, there are some exceptions. Relational operators function globally, so that the expression  $A > B > C$  means the same as  $(A > B) \text{**} (B > C)$ .

**Table 4-3. The Hierarchy of Operators in MASM**

Level	Operators	Description
0	->	Conditional if-then
	!	Conditional else (exclamation point)
	*	Set leading asterisk flag (unary,prefix)
	/	Control information context (unary,prefix)
1	\	Negation operator (NOT) (unary,prefix)
2	<	Less than
	<=	Less than or equal
	>	Greater than
	>=	Greater than or equal
	=	Equal
	<>	Not equal
	==	Node identity
	!=	Node nonidentity
3	:	String concatenation
4	++	Bit logical OR
	--	Bit logical XOR
5	**	Bit logical AND
6	+	Arithmetic addition
	-	Arithmetic subtraction
7	*	Arithmetic multiplication
	/	Arithmetic division quotient
	//	Arithmetic division covered quotient
	///	Arithmetic division remainder
8	*/	Fixed-point integer scaling
	*+	Floating-point power of 10 positive scaling

continued



**Table 4-3. The Hierarchy of Operators in MASM (cont.)**

Level	Operators	Description
	*-	Floating-point power of 10 negative scaling
	*//	Floating-point power of 2 scaling
9	+	Positive number (unary,prefix)
	-	Arithmetic negative (unary,prefix)
10	D	Double-precision (unary, postfix)
	S	Single-precision (unary, postfix)
	L	Left-justify space-fill (unary, postfix)
	R	Right-justify zero-fill (unary, postfix)

#### 4.3.2.2. Conversion Rules

Some operators demand a particular form for their operands. In some cases a transfer function is automatically called to ensure that the operands are the proper type. Integers are converted to floating-point if used in mixed mode, and similarly, strings are converted to integers.

Certain transfer functions are not defined, such as those converting relocatable values to floating-point. When mixed-mode arithmetic of this type is attempted, an R flag is generated.

Relocatable values cannot be operated on by logical operators or string operators. Also, they cannot be multiplied by a value other than nonnegative integers, or combined with floating-point numbers. Violation of these restrictions produces an R flag and relocation is lost.

### 4.4. Data Types

MASM has a large number of functions that allow the programmer to interrogate the data type of an expression.

#### 4.4.1. \$TYPE(*e*) — Compute Data Type Number

The expression *e* is converted according to the rules for parameter conversion. The value of \$TYPE is an integer corresponding to the type of the expression *e* as given by Table 4-4.

**Table 4-4. Data Type Numbers**

Number	Data Type
1	Integer value.
2	Floating-point value.
3	String value.
4	Node reference.
5	Internal name (NAME line label).
6	PROC name (label on PROC line).
7	FUNC name (label on FUNC line).
8	MASM directive (including instruction mnemonics and forms).
9	MASM built-in function.

#### 4.4.2. Type Testing Functions

All the functions described in this subsection require one parameter converted according to the rules of parameter conversion. They all return either 0 or 1, depending on the similarity between the type of the parameter and the function used.

Table 4-5 lists the type testing functions provided by MASM.

**Table 4-5. Description of Type Testing Functions**

Function Name	Data Type (Value = 1)	Description
\$TBIN	1	Test for integer
\$TCON	5,6,7,8,9	Test for control information
\$TDAT	1,2,3,4	Test for data
\$TDIR	8	Test for directive
\$TFLT	2	Test for floating-point
\$TFNM	7	Test for a FUNC name
\$TFUN	9	Test for a built-in function
\$TINM	5	Test for an internal name
\$TNAM	5,6,7	Test for a name
\$TNOD	4	Test for a node
\$TPNM	6	Test for a procedure name
\$TSTR	3	Test for a string
\$TVAL	1,2,3	Test for a value

### 4.4.3. \$IBITS(*e*) — Indicator Bits for Expression

The expression *e* is converted according to the rules for parameter conversion. Table 4-6 indicates the bits set in the result for various characteristics of *e*.

**Table 4-6. Expression Characteristics Indicator Bits**

Bit	Meaning
0*	Flagged expression
1	Double precision
2	Negative arithmetic value
3	Left justification

continued

\* least significant bit

**Table 4-6. Expression Characteristics Indicator Bits** (cont.)

Bit	Meaning
4	Form attached
5	Fielddata string
6	ASCII string
7	Redefined value
8	Transformed negative value

\* least significant bit

For example, the value of \$IBITS(\*'ABC'LD) is 0113 if the current character representation mode is ASCII.

# Section 5

## Data Generation

A MASM statement with a void operations field and a nonvoid operand field generates data that is sent to the relocatable binary element. MASM accomplishes this by an implied call to the \$GEN directive (see 5.3).

The operand field format is as follows:

$$e_1, e_2, \dots, e_n$$

where  $e$  is any valid MASM expression. Each  $e$  represents a subfield (see 2.3.1). If the number of subfields,  $n$ , is a divisor of the word size,  $b$ , an  $n$ -field word is generated with each field  $b/n$  bits in size. If  $n$  is not a divisor of the word size, then an E flag is generated.

Each  $e$  can have an unary plus or minus sign (+ or -) or the entire operand can have a unary plus or minus. If the entire operand has a sign, the operand field must be enclosed in parentheses preceded by the desired sign.

### Example 1

```
$WRD 36  
+ 15
```

This example generates one 36-bit word with the value 15. The generated octal output appears as 000000000017.

### Example 2

```
$WRD 32  
+(-4,5,6,7)
```

This example generates a 32-bit word consisting of four 8-bit fields. The generated octal output appears as 373 005 006 007 if broken down into its fields; or 37301203007 as a 32-bit word.

### Example 3

```
$WRD 16  
+ 3,2,1,0,1,2,3,2
```

This example generates a 16-bit word consisting of eight 2-bit fields.

### 5.1. Signed Character Strings

If the character string is signed, the data generated is right-justified, zero-filled. The number of words generated,  $n$ , varies according to the following relationship:

$$n = (c * b) / k$$

where:

$c$

is the number of characters.

$b$

is the number of bits per character.

$k$

is the number of bits per word.

If the value of  $n$  is larger than 2, a T flag is generated. If the word size is greater than 36 bits,  $n$  cannot be larger than 1.

#### Example 1

```
$INSERT      ' WRD 36', ' $FDATA'
```

This instruction sets the word size to 36 bits and the character set to Fieldata. (See 6.6.)

#### Example 2

```
+ 'ABCEF'
```

MASM generates the following value:

```
000607101213
```

#### Example 3

```
- 'ABCEFGH'
```

MASM generates the following octal value:

```
77777777771  
706765646362
```

## 5.2. Unsigned Character Strings

If the operand field is a string enclosed in single quotation marks, without a leading sign, a variable number of data words can be generated. The value generated is a left-justified, space-filled set of words. The number of words generated is determined by the number of characters in the character string and the system character set. Thus, more than two words can be generated.

### Example

```
1.      $FDATA
2.      'ABCD'
3.      'ABCDEFGH'
4.      $ASCII
5.      'ABCD'
```

### Explanation

Line 1

Sets the system character set to Fieldata.

Line 2

Generates the following octal value:

```
060710110505
```

Line 3

Generates the following octal value:

```
06710111213
150505050505
```

Line 4

Sets the system character set to ASCII.

Line 5

Generates the following octal value:

```
101102103104
```

### 5.3. \$GEN — Data Generation

The \$GEN directive has the following format:

\$GEN  $e_1, \dots, e_n$

where the action depends on the number of operands. If only one operand is present, its value is generated as data for the output element. When  $n$  is greater than 1, the current word size is divided into  $n$  equal fields, each  $e_i$  is converted to a binary value, and the value of  $e_i$  is generated in the  $i^{\text{th}}$  field. The \$GEN directive is called implicitly for lines which have no operation field or a void operation field.



# 5.4. Forms

This subsection describes the MASM directives available to tailor the generated data.

## 5.4.1. \$FORM — Define a Form

The format of the \$FORM directive is as follows:

```
label      $FORM      e1,e2,...,en
```

where  $e_1,...,e_n$  are integer values greater than zero, and the sum of these values is less than 73. The integer values in the operand field represent the length in bits of the field of a word (or double word).

The label used on the \$FORM is defined as a form name, which can be used in the operation field of a MASM line to specify generation of a datum. The label is used as a form reference as follows:

```
label      d1,d2,...,dn
```

Each  $d_i$  is converted to a binary value and mapped into a field of size  $e_i$ , as specified on the \$FORM line defining the form name. The fields of a form are adjacent and right-justified within a word or double word.

Forms must be defined before they are used.

### Example 1

```
1.      PF      $FORM      12,6,18
2.      PF      5,1,TAG
```

### Explanation 1

Line 1

The symbol PF is the form name and is associated with a 36-bit word divided into three fields of 12, 6, and 18 bits.

Line 2

This is a form reference line that produces a 36-bit word with the values 5,1,TAG in the fields defined by the symbol PF. If the symbol TAG was associated with the value 01000, then the octal representation is as follows:

```
000501001000
```

A field in a form reference can be a line item. If the form of the line item is identical to the form referenced and is not a literal, the operator OR is applied to the corresponding fields of both forms referenced.

### Example 2

```
1.      FA      $FORM      12,6,18
2.      FB      $FORM      12,6,18
3.      S1      $EQU       +(FA 0,1,TAG)
4.      FB      4,S1,0
```

### Explanation 2

#### Line 1

A symbol is defined having an associated form.

#### Line 2

A symbol is defined having an associated form.

#### Line 3

A line item is created with one of these forms, FA. If the symbol TAG has the value 01000, then the octal representation is as follows:

```
000001001000
```

The plus sign preceding the line item inhibits literal generation.

#### Line 4

A form reference line using S1 as one of the values can be represented in octal as follows:

```
000401001000
```

The result is produced as follows:

```
S1      =      000001001000
FB      =      000400000000
result  =      000401001000
```

## 5.4.2. \$GFORM — Generalized Form

The format of the \$GFORM directive is as follows:

```
$GFORM       $f_1, e_1, f_2, e_2, \dots, f_n, e_n$ 
```

where  $f_i$  and  $e_i$  are all converted to binary values, and  $f_1, \dots, f_n$  must be unrelocated positive integers whose sum is less than 73. The \$GFORM directive provides the same effect as the following lines:

```
F      $FORM       $f_1, \dots, f_n$ 
F      F           $e_1, \dots, e_n$ 
```

without actually creating the form F. If any  $f_i$  is 0, the corresponding  $e_i$  is ignored after conversion to binary.

## 5.5. Location Counter Specification

MASM allocates storage for instructions and data under the control of storage location counters. There are 64 location counters available in MASM, numbered 0 through 63. Any location counter can be used or referenced in any sequence. The initial location counter number is 0. A program remains under control of that location counter number until a new location counter specification is introduced. When a location counter is specified, it controls all subsequent coding until another location counter is explicitly specified.

### 5.5.1. \$(e) — Location Counter Value

The \$(e) function is the same as \$LCV when used as an expression element. When written in the label field of a line (the argument is mandatory), this function indicates a change in the number of the active location counter to the value of *e*. The data that follows is then generated under location counter *e*. This is the only built-in function that can be written in the label field of a line; therefore, built-in functions cannot be redefined.

### 5.5.2. \$RES — Reserve Space

The \$RES directive has the following format:

```
$RES e
```

where *e* is a binary value without relocation. If the current location counter is blocked, this line is marked with an I flag and no action is taken. Otherwise, the value of *e* is added to the current location counter. If the directive appears on a source image, the original value of the location counter is printed in the listing.

**Note:** *The expression e must be fully computable in the summary pass of the subassembly, since the location counter value is affected. This means that any identifiers used in computing e must have their values determined by previously interpreted lines.*

Location counter offsets cannot be longer than 262,143 when generating a relocatable, or longer than 16,777,215 when generating an object module. A T flag is generated when *e* is greater than the allowed offset.

### 5.5.3. \$LCN — Location Counter Number

The \$LCN function requires no parameters. Its value is the number of the current location counter.

### 5.5.4. \$ILCN — Initial Location Counter Number

The \$ILCN function requires no parameters and returns the location counter number that is in effect at the beginning of the current subassembly pass. For the main assembly, the value is always zero. If the current subassembly is a call to a procedure with a specified location counter number, then \$ILCN is the value of that location counter number. For other subassemblies, \$ILCN is the value of \$LCN at the point at which the subassembly was invoked.

#### Example 1

Inside a procedure started by the following line, the value of \$ILCN is 5.

```
P      PROC      , , 5
```

#### Example 2

Inside a procedure started by the following line, called from a point where the active location counter is 3, the value of \$ILCN is 3.

```
P      PROC      .
```

### 5.5.5. \$LCV(*e*) — Location Counter Value

The \$LCV function returns the current value of the location counter designated by the value of *e*. If *e* is omitted, the value of \$LCN is used for *e*. This function has the \$ function as a synonym for compatibility with existing programs and for shorthand convenience.

### 5.5.6. \$LCB(*e*) — Location Counter Base

The \$LCB function requires one parameter or no parameters. The value of *e* should be in the range 0 to 63, if given. If *e* is not within this range, a T flag is produced. If no parameter is given, the value of \$LCN is used. The value returned by \$LCB is 0, relocated by the location counter specified by the value of *e*. In other words, the value of \$LCB is the address of the first word of location counter *e*.

### 5.5.7. \$LCFV(*e*) — Location Counter Final Value

The \$LCFV function requires one or no parameters. The value of *e* should be in the range of 0 to 63, if given. If *e* is not within this range, a T flag is produced. If no parameter is given, the value of \$LCN is used for *e*. The value returned by \$LCFV is the final value of the location counter at the end of the summary pass, without relocation. If the location counter has not been referenced, then \$LCFV returns a value of -1.

The \$LCFV function returns a value of zero under the following conditions:

- Blocked location counter.
- Location counter referenced but no words generated.
- Location counter referenced but only literals generated. Since literals are only generated on the generative pass of the assembler, they cannot be accounted for by the \$LCFV function.

### 5.6. \$LIT — Literal Pool Definition

The \$LIT directive has the following forms:

```
label      $LIT  
            $LIT
```

If the current subassembly pass is not generative, the directive is ignored. If there is no label, the implied literal pool counter number is set equal to the current location counter number. If there is a label field, then a literal function is created that places literals into the pool corresponding to the current location counter, and the label is set equal to that function. At the beginning of the main assembly, the implied literal pool location counter number is 0. If there is more than one labeled \$LIT directive for the same location counter, the labels are the same function; that is, literal pools are unique by location counter only, not by name.

For example, if the following lines are interpreted,

```
$(2),ABC      $LIT  
$(4)          $LIT  
$(1)          (1,TABLE)  
              ABC(1,0)
```

then the literal (1,TABLE) is placed in the location counter 4 literal pool, while the literal ABC(1,0) is placed in the location counter 2 literal pool.

### 5.7. \$WRD — Specify Word Size

The \$WRD directive has the following format:

```
$WRD      e
```

where *e* is a positive binary value, without relocation, not exceeding 36. The current word size in bits is set to the value of *e*. Internally, MASM can handle a word size up to 72 bits; however, the present operating system interface relocatable output routine (ROR) does not support a word size larger than 36 bits.

## 5.8. \$NEG — Transform Negative Values

The \$NEG directive has the following format:

```
$NEG      function-name
```

where *function-name* is an entry point to a user-defined function, called internally by MASM to transform negative values. If the function name is void, the effect of the \$NEG directive is nullified. The function is called in the following instances:

- When a negative value is being output to the relocatable binary element.
- When a negative value that is part of a larger value is being built. This applies to both explicit and implicit forms. This means the transformed value can be entered into the dictionary.

The \$NEG directive and its associated function is in effect for all lines following it or until another \$NEG directive is encountered.

MASM supplies the following parameters to the user-defined function:

- The value as selector 1
- The field size of the value in bits as selector 2

### Example 1

This example shows two's complement transformation.

```

1.      @MASM,S MASM*MSM4.NE001,TPF$.
2.      MASM xxRyy
3.
4.
5.
6.
7.      777776 777775
8.
9. 0 000000 175 17766 0004 36
10.     777777777775
11. 000001 777777777776
12. 000002 777777777766
13.
14.
15. LOCATION COUNTERS:  $(0) 000003  $(1) 000000
16. END MASM - LINES: 31 TIME: 4.727 STORAGE: 8028

```

```

1. f$      $func
2. twos*    $name
3.          $end f$(1)=-1->-0!f$(1)=-0->+0!f$(1)+1
4.          $neg      twos
5. az       $equ      +(-2,-3)
6. af       $form      7,13,11,5
7.          af        -3,-10,4,-2
8. ax       $equ      -2
9.          +          ax
10.         -          10
11.          $end

```

### Explanation 1

Line 3-5

Defines the function MASM uses to perform the transformation of negative values from one's complement to two's complement.

Line 6

Indicates that the function TWOS transforms negative values.

## Data Generation

---

### Line 7

Associates the transformed value 077777677775 with the symbol AZ and enters them in the dictionary.

### Line 8

Defines the symbol AF as a form reference.

### Line 9

Uses the form reference AF to generate the 36-bit value 0767775400236 or, broken down into its input fields, 0175 017766 0004 036.

### Line 10

Associates the value -2 (one's complement) with the symbol AX and enters them in the dictionary.

### Line 11

Places the value associated with the symbol AX into the relocatable binary element, after the value has been transformed to a two's complement negative value.

### Line 12

Transforms the value -10 to a two's complement number and places the value into the relocatable binary element.

Be careful when performing arithmetic operations on values that consist wholly or partially of values that were transformed.

## Example 2

This example shows sign bit magnitude transformation.

```
1. @MASM,S MASM*MSM4.NE002,TPF$.
2. MASM xxRyy
3.
4.
5.
6.
7.
8.          77777777772
9. 0 000000 400000000005
10. 000001 400000000012
11.
12.
13. LOCATION COUNTERS: $(0) 000002 $(1) 000000
14. END MASM - LINES: 22 TIME: 2.303 STORAGE: 28918

1. F$          $FUNC
2. SBMAG*      $NAME
3. MSK         $EQU F$(2)>36->(1*/(F$(2)-1))D!1*/(F$(2)-1)
4.             $END (-F$(1))+MSK
5.             $NEG SBMAG
6. SB          $EQU 5
7.             + SB
8.             - 10
9.             $END
```

## Explanation 2

### Line 3-6

Defines a function that transforms one's complement negative values to sign bit magnitude negative values.



Line 5

Determines the location of the sign bit from the field size of the value as passed in selector 2.

Line 6

Complements the value specified by F\$(1) for the magnitude, and then logically ORs the result with the sign bit provided by MSK.

Line 7

Indicates that the function SBMAG transforms negative values.

Line 8

Associates the value -5 (one's complement) with the symbol SB, and enters them in the dictionary.

Line 9

Places the value associated with the symbol SB into the relocatable binary element after the value has been transformed to a sign bit magnitude negative value. The generated value is 0400000000005.

Line 10

Transforms the value -10 to a sign bit magnitude negative value, and places the value into the relocatable binary element. The generated value is 0400000000012.

As mentioned under the two's complement transformation, you should be careful when performing arithmetic operations on values that consist wholly or partially of transformed values.



# Section 6

## Assembly-Time Controls

This section describes directives that control an assembly at assembly-time. This section includes the following topics:

- Directives for handling conditionally assembled code
- Looping directives
- No operation directive
- Transferring control
- Defining internal names
- Source insertions
- Procedures and functions (creating, accessing, using, and manipulating)
- Microstrings
- Levelers

### 6.1. Condition Testing Directives

The directives in this subsection allow the user to conditionally generate MASM instructions.

#### 6.1.1. \$IF — Conditional Interpretation

The format of the \$IF directive is as follows:

```
$IF      e
```

where *e* is an integer value without relocation. If *e* is omitted, a value of 0 is used. \$IF increments the conditional nesting level by 1. If MASM is already skipping images for an outer conditional construction, this action continues. If not, the expression *e* is evaluated and compared with 0. If *e* is not equal to 0, MASM continues to interpret images. If *e* is 0, MASM begins to skip images and continues to do so until a matching \$ELSE, \$ELSF, or \$ENDF is found.

\$IF has the synonym ON.

### 6.1.2. **\$ELSE — Conditional Interpretation Alternative**

The format of the \$ELSE directive is as follows:

```
$ELSE
```

The \$ELSE directive requires no parameters. It is used with the \$IF and \$ENDF directives to establish an alternate set of code to be interpreted. If MASM has been conditionally skipping images when \$ELSE is encountered, skipping discontinues and interpretation begins. If MASM has been conditionally interpreting images when it encounters the \$ELSE directive, MASM discontinues interpretation and begins skipping.

### 6.1.3. **\$ENDF — End Conditional Interpretation Group**

The format of the \$ENDF directive is as follows:

```
$ENDF
```

This directive ends the conditional code generation group introduced by the most recent use of the \$IF directive. It requires no parameters. Conditional interpretation or skipping for this group stops, and the mode of interpretation reverts to that effective for the next outer level of \$IF - \$ENDF, if any.

\$ENDF has the synonym OFF.

### 6.1.4. **\$ELSF — Conditional Interpretation Conditional Alternative**

The format of the \$ELSF directive is as follows:

```
$ELSF    e
```

where *e* is an integer value without relocation. Use this directive with the \$IF-\$ENDF directives. This directive operates as an \$ELSE directive if MASM is already interpreting images. If MASM has been skipping images when it encounters this directive, MASM evaluates the expression and either interprets or skips the images that follow it.

### Example 1

```
$IF      A
.
.
a
.
.
$ELSF    B
.
.
b
.
.
$ENDF
```

### Example 2

```
$IF      A
.
.
a
.
.
$ELSE
$IF      B
.
.
b
.
.
$ENDF
$ENDF
```

### Explanation

Examples 1 and 2 are logically the same; however, example 1 is shorter.

### 6.1.5. \$ANDF — And If

The format of the \$ANDF directive is as follows:

```
$ANDF      e
```

where *e* is an integer value with no relocation. Use this directive with the \$IF, \$ELSE, \$ENDF, and \$ELSF directives. This directive is ignored if MASM is already skipping images within a conditional construction. If not, *e* is evaluated. If *e* is nonzero, no action is taken; if *e* is 0, MASM begins skipping images.

#### Example 1

```
$ANDF      e  
.  
.  
.  
d
```

#### Example 2

```
$IF        e  
.  
.  
.  
$ENDF  
d
```

#### Explanation

In the preceding examples, *d* denotes one of the directives \$ENDF, \$ELSE, or \$ELSF. Example 1 is equivalent to (and shorter than) example 2.

### Example 3

```
1.  $IF      A>0
2.  +        A
3.  $ANDF    B>0
4.  +        B
5.  $ENDF
```

### Example 4

```
1.  $IF      A>0
2.  +        A
3.  $IF      B>0
4.  +        B
5.  $ENDF
6.  $ENDF
```

### Explanation

Examples 3 and 4 are functionally equivalent.

If the expression  $A>0$  on line 1 is true, then the statements on lines 2 and 3 are interpreted. If  $A>0$  is false, then lines 2 through 4 are skipped.

If line 3 is interpreted and the expression  $B>0$  is true, then line 4 is interpreted. If the expression  $B>0$  is false, then line 4 is skipped.

The `$ENDF` in line 5 marks the end of the conditional construct.

## 6.2. Looping Directives

The directives in this subsection allow the user to conditionally and repetitively generate MASM instructions.

### 6.2.1. \$DO — Repetitive Generation of a Line

The format of the \$DO directive is as follows:

```
[label]    $DO    rpt ,line
```

where *label* is optional and can be any identifier (selections not permitted), *line* is any valid MASM line (excluding leveler and page ejector), and *rpt* is from one to three integer values separated by commas. At least one space must occur between *rpt* and the following comma, and if the line to be generated has a label, the label must immediately follow the comma. If *rpt* is only one value, it is interpreted as *end*; two values are interpreted as *start* and *end*; while three values are interpreted as *start*, *end*, and *step*. If *step* or *start* are not specified, values of 1 are used. The *label* is set to the value *start* and incremented by *step* until the value *end* is reached or passed. For each value given to the label, the specified line is interpreted once.

The *start* and *end* values must be positive integer values without relocation and cannot exceed 262143. The *step* field can be positive or negative but not zero, and it cannot exceed 131071 in magnitude. If  $(end-start)/step$  is negative, the line is interpreted zero times, and the label is not defined. Any microstrings in the object line are interpreted each time the line is interpreted; microstrings preceding the space-comma pair are interpreted once (before initiating the \$DO). \$DO directives can be nested. A \$DO repetition can be terminated by the \$ENDD directive before the full number of repetitions are performed.

### 6.2.2. \$ENDD — End \$DO Iteration

The format of the \$ENDD directive is as follows:

```
$ENDD
```

This directive requires no parameters and can be used only if a \$DO repetition is being performed. The \$ENDD terminates the current active group of nested \$DO repetitions.



### 6.2.3. \$REPEAT — Repeat a Statement Group

The \$REPEAT directive has the following form:

```
label      $REPEAT      rpt
```

where *label*, if present, cannot have selectors, and *rpt* is a field of zero to three integer expressions without relocation. If there is one expression in *rpt*, it is assumed to be *end*; two expressions are assumed to be *start* and *end*; and three expressions are assumed to be *start*, *end*, and *step*. If *start* or *step* is omitted, the value 1 is used. If *end* is omitted, the value 262,143 is used. Both *start* and *end* must be in the range 0 to 262,143, while *step* must be nonzero with magnitude less than 131,072.

The lines between \$REPEAT and the next (unconditional) \$ENDR at the same \$REPEAT nesting level are saved as a sample. If *step* does not have the same sign as *end-start*, the sample statements are not interpreted. Otherwise, the sample statements are interpreted  $1 + (\text{end} - \text{start}) / \text{step}$  times, with the label set first to *start*, then to *start+step*, and so on. Each iteration is terminated by encountering an \$ENDR directive, either the unconditional one at the end of the sample, or one generated conditionally. If an \$ENDI directive is encountered, the entire \$REPEAT operation terminates. \$REPEAT introduces a new sample level but not a new dictionary level.

### 6.2.4. \$ENDR — End \$REPEAT Construction

The format of the \$ENDR directive is as follows:

```
$ENDR
```

This directive requires no parameters and cannot have a leveler or label field because it is not saved as part of \$REPEAT sample.

This directive has two functions. First, when a \$REPEAT group sample is being picked up before starting iteration, the directive indicates the end of a \$REPEAT group construction, and therefore cannot be conditionally generated when used this way. When \$ENDR is encountered during repetition, either at the end of the \$REPEAT group or through its conditional generation, the current iteration ends, the counter is incremented, and the next iteration begins.

### 6.2.5. \$ENDI — End \$REPEAT Iteration

The format of the \$ENDI directive is as follows:

```
$ENDI
```

The \$ENDI directive requires no parameters and can be used only if a \$REPEAT construction is active. This directive terminates the iteration of the currently active \$REPEAT group, thus giving control to the next outer \$REPEAT group, if any, or to the main assembly.

## 6.3. \$NIL — No Action

The format of the \$NIL directive is as follows:

```
$NIL
```

The \$NIL directive requires no parameters and generates no code. If a label (including a waiting label) is specified, the label is marked as used (preventing definition by implication) but is not given a value and is not entered into the dictionary.

## 6.4. \$GO — Transfer to a Name

The format of the \$GO directive is as follows:

```
$GO      n
```

where *n* must evaluate to an internal name (a label that appears on a \$NAME directive). If the \$GO directive is in the main assembly, only forward transfer is possible; MASM begins skipping images until the specified name is encountered. If the \$GO directive is in a procedure or function, transfer can be made to any name of that procedure or function, or any external name of any other function or procedure. If the transfer is out of the present function or procedure, a diagnostic G-flag is produced. Such transfers are lateral transfers and do not change the subassembly nesting level. A forward \$GO directive within a procedure/function to a nonexternalized name is done by skipping images and can thus be slower than other \$GO operations. To terminate the present procedure or function interpretation, it is not necessary to do a \$GO to a name immediately before the \$END directive at the end of the sample. The following alternatives are preferable:

DO 1 , END or 1->END using conditional operators

DO *e* , END or *e*->END using conditional operators

The first of these is an unconditional termination, while the second is conditioned on the value of the expression *e*.

### 6.5. \$NAME — Define an Internal Name

The format of the \$NAME directive is as follows:

```
label      $NAME      e
```

where *e*, which can be void, is converted according to the rules for parameter conversion. The label specified is given the value of an internal name, whose associated entry value is the value of *e*. An internal name can be used to provide an alternate entry point to a procedure or function, or it can provide a forward transfer point within the main assembly. The object of a \$GO directive must be an internal name. For an internal name to be known outside the procedure or function containing it, the name must be externalized. Nonexternalized names can be used only within the procedure or function containing them (or deeper nested calls). They are generally used only as \$GO destinations. For \$NAME directives contained within a procedure or function sample, the expression *e* is evaluated only once, at the time the sample is initially scanned. This means that the value of *e* does not change from call to call. The value of *e* is obtainable as P(0,0) or F(0), assuming that P is the relevant \$PROC label or F the relevant function label.

### 6.6. \$INSERT — Insert Images

The format of the \$INSERT directive is as follows:

```
$INSERT      e1, . . . , en
```

where *e*<sub>1</sub>, . . . , *e*<sub>*n*</sub> are strings in a system character set. Each *e*<sub>*i*</sub> is treated as a line interpreted by MASM, beginning with a label field as the first character of the string and continuing with the rest of the line. The lines defined by the strings *e*<sub>*i*</sub> are interpreted in order from left to right. \$INSERT directives can be nested. Lines interpreted at inner levels are interpreted at the proper place between lines interpreted at higher levels.

#### Example

```
$INSERT      'RPT JGD R4,TOP', ' J EXIT'
```

#### Explanation

This has the same effect as the following lines:

```
RPT      JGD      R4,TOP
J        EXIT
```

## 6.7. Procedures and Functions

This subsection describes the following topics:

- Defining procedures and functions
- Calling procedures and functions
- Nesting procedures
- Types of procedures

### 6.7.1. Procedures

Procedures are one way of calling separate subassemblies within the main assembly. You can use these subassemblies in the following ways:

- Extend the set of directives and instruction mnemonics provided by MASM
- Build data structures
- Generate sequences of coding or data

Procedures can use the full capabilities of MASM, with the restrictions mentioned on each type of procedure and those restrictions caused by the context of the call (for example, a line item).

A procedure is bounded by a \$PROC-\$END pair of directives. The lines between the \$PROC and \$END directives are called the procedure sample, and MASM interprets them when the procedure is invoked. The \$END directive that terminates the body of the procedure must be unconditional (that is, not the object of a \$DO, within an \$IF-\$ENDIF pair; or created by an \$INSERT directive or microstring substitution).

When supplying expressions on the \$PROC directive, a conditional expression can generate a void subfield and has a different meaning from a subfield whose value is zero. This is true for all three subfields. Since MASM interprets the \$PROC directive when it encounters it and does not save the directive as part of the procedure body, MASM evaluates the expressions in the \$PROC operand field only once, when it saves the procedure sample. The label on the \$PROC directive line, however, is defined at the level of the body of the procedure and must be externalized if it is to provide an entry point to the procedure. The same is true for the label on any \$NAME lines inside the procedure; MASM evaluates \$NAME line expressions only once, at the time it saves the procedure sample. (If the body of the procedure is to be computed by actions taken at the time sample is saved, use levelers as described in 6.9.) Terminate a procedure sample by an unconditional, nongenerated \$END directive. Procedure interpretation terminates when MASM encounters any \$END directive, either one generated conditionally or the one at the end of the procedure body.

Generally, procedures generate sequences of coding or data that vary based on some set of parameters that can be either explicitly given to the procedure on the procedure call line or implicitly determined from values defined at higher levels. Procedures can generate any number of words of data or instructions, including zero. However, procedures must consistently increment location counters from pass to pass of the

higher-level subassemblies. If the incrementation is not done, the output is likely to contain C and D flags that indicate that the values of the location counters were different in different passes.

Procedures must be defined before they are called. You can define procedures in one of the following ways:

- The procedure sample can be included in the source language given to MASM.
- The procedure sample can be contained in the dictionary information saved by a definition mode assembly and loaded by the \$INCLUDE directive (see 12.3).
- The procedure sample can be in a file in an assembler procedure element processed by the procedure definition processor (PDP).

The rules for procedures name lookup in files are in NO TAG. See also the *Procedure Definition Processor (PDP) Operations Reference Manual*.

### 6.7.1.1. \$PROC — Define a Procedure

The \$PROC directive introduces a procedure definition. The format of this directive is as follows:

```
[label]    $PROC    [e1[,e2[,e3]]] [e4[,e5]]
```

Parameters  $e_1$ ,  $e_2$ , and  $e_3$  are nonnegative integer values without relocation.

The optional label field cannot have selectors. The label is used within the procedure to identify the parameter tree defined by the call to the procedure. If the label is externalized, it is also given the value of an entry point to the procedure (at the appropriate level), with no entry parameter that enters the procedure at the first statement.

The parameter  $e_1$  specifies the maximum number of parameter lists allowed (in addition to list 0). If  $e_1$  is void, then the number of lists is unlimited. If  $e_1$  is flagged, the procedure is defined as one pass (see 6.7.8).

The parameter  $e_2$  specifies the number of words generated by the procedure. This value is computed only once, when the sample is scanned, and it cannot be changed from call to call. Such a procedure is called a words-given procedure (see 6.7.8.3).

The parameter  $e_3$  specifies the location counter used for generation of the code under the procedure. If omitted or void, the location counter used is the one active at the point the procedure is called.

Parameters  $e_4$  and  $e_5$  are optional and normally are used internally by Unisys. These parameters are for control of basic and extended mode generation and for indicating 18-bit addressing, respectively. Parameters  $e_4$  and  $e_5$  are defined in Appendix A (see A.2.10).

### 6.7.1.2. \$END — End of a Subassembly

The format for the \$END directive as follows:

```
$END    e
```

*e* is converted according to the rules for parameter conversion. The \$END directive terminates a subassembly and ends a procedure or function sample. When used to terminate a procedure or function sample, the \$END directive cannot be conditionally generated. When a sample is being interpreted, however, an \$END directive can be generated conditionally and thus terminate execution of that procedure or function.

For a procedure, the expression *e* is ignored if present. For a function, the value of *e* is returned as the value of the function call, and *e* can have any value, including strings, nodes, or control information. If *e* is present on the \$END directive that terminates the main assembly, the value of *e* must be an integer with exactly one relocation item (which is not an external reference). This indicates to the Collector (and ultimately to the Executive) the address at which the execution of the generated absolute program begins. If more than one element in a collection has a transfer address *e* specified by a main assembly \$END directive, ambiguity exists that must be resolved by Collector source language. An element having a defined starting transfer address is called a main program. For further details, refer to the ER Programming Reference Manual.

### 6.7.1.3. Calling a Procedure

To call a procedure, write the name of an entry point to the procedure in the operation field of a MASM line or line item. Procedure entry points are created by externalization of labels on \$PROC directives and \$NAME directives, or as the value returned by a call on the \$FN built-in function. The operand fields (and possibly subfields) are passed to the procedure as parameters by creating a new node whose selections are defined as follows:

(0,0)

Procedure entry parameter (value of expression on \$NAME line if entry made by \$NAME label, or value specified by \$FN built-in function). Not defined if entry is made by \$PROC label.

(0,j)

If (0,0) is a defined selection, this is the value of the expression in the  $j^{\text{th}}$  subfield of the operation field, numbering the procedure entry name itself as 0.

(i,j)

This is the value of the expression in the  $j^{\text{th}}$  subfield of the  $i^{\text{th}}$  operand field. Operand fields and subfields are numbered consecutively, beginning with 1.

(i,\*j)

This is 1 if the expression in the  $j^{\text{th}}$  subfield of the  $i^{\text{th}}$  operand field was flagged; otherwise, it is zero. The value of  $i$  can be zero, if (0,0) is a defined selector. This retrieves the flag attribute of the \$NAME line expression if  $j = 0$ .

The node for which these selections are defined is given as the local definition of the label that appeared on the \$PROC directive line. Thus, if the procedure began with the following line

```
P      PROC      ...
```

the value of the third subfield of the first operand field is obtainable as P(1,3), while a name entry parameter is retrieved by P(0,0).

The parameters in the operand (and operation) subfields of a procedure call are converted according to the rules for parameter conversion. Control information is possible, thus allowing P(1,1) to appear in the operation field of a line within the procedure.

### 6.7.2. Functions

Functions provide a way to extend the set of functions provided by MASM as built-in functions with new user-created functions. Functions can use the full capabilities of MASM, but are generally restricted in the generation of data because the current location counter is blocked.

A function is delimited by a \$FUNC-\$END pair of directives, where the \$END terminating the text of the \$FUNC directive must be unconditional (that is, not the object of a \$DO directive, not within a \$IF-\$ENDIF pair, not created by \$INSERT or other conditional construction). The lines between the \$FUNC and \$END directives are called the function sample, and MASM interprets them when the function is called. When MASM encounters the first \$END image, interpretation of the function is terminated whether or not this \$END was conditionally generated. The value returned by the function is given by an expression in the first operand field of the \$END directive that terminates the function interpretation.

A label is generally present on the \$FUNC directive line. When interpreting the function sample, this label is given the value (local to the function subassembly) of a node reference with either  $n$  or  $n_{+1}$  selectors defined. The value of selector 1 is that of argument 1 of the function call. Selector 2 is given the value of argument 2, and so on up to argument  $n$  for selector  $n$ .



If the function is entered by a name line, the expression on the name line (evaluated when the function sample was first scanned) is used as the value of selector 0 of the function name node reference.

Outside the function, the value of the label on the \$FUNC directive line is known only if the label was externalized. The label is then an entry point value of the function with no entry parameter. This value is control information, so most uses of the value in expressions where a function call is not intended require the use of the control information operator (/).

Similarly, any name line used to provide an entry to the function must have its label externalized, and such a label is control information. Name line labels can be used as \$GO directive objects within the function. The \$NAME similarity between the function parameter and entry mechanism and that for procedures should be evident.

**Note:** *The procedure parameter tree has two levels of selection.*

Functions can call other functions and procedures. Any MASM operation is permitted within a function or in any function or procedure called from a function, except when a blocked location counter is being incremented. Since a procedure can generate code under an unblocked location counter and then call a function, more than one location counter can be blocked at a given time.

Functions, like procedures, are subassemblies. They introduce a new dictionary level for local definitions. The lookup and definition of local and external symbols is the same as for procedures, except that a function cannot have a waiting label. On the other hand, a function is never processed twice by the same higher-level subassembly. Since functions normally do not generate data, a generative pass is not needed. Therefore, forward references cannot occur. A function can be computed on either the summary or generative pass of the next higher subassembly.

### Example 1

This function calculates the character position in the first argument of the first substring that is equal to the second argument. Both arguments are assumed to be strings.

```

1.      F      FUNC
2.      INDEX*  NAME
3.      K      REPEAT      $SL(F(1))-$SL(F(2))+1
4.      IF      F(2)=$SS(F(1),K,$SL(F(2)))
5.      ENDI
6.      ENDF
7.      ENDR
8.      END      K

```

K is incremented until the ENDI directive is interpreted. The value of K is available outside the REPEAT group, and it is returned by the function as its value. If the substring is not found, this function still returns a value for an index, even though it is incorrect. Normal coding custom for this procedure would return a 0 value in the no-find case. Thus the value of INDEX('ABCABCABC','BCA') is 2, but the value of INDEX('ABCABCABC','E') is 9.

### Example 2

A function that converts a character string into a node reference whose selector *i* is a single character string containing the *i*th character of the argument string can be written as follows:

```
1      F      FUNC
2      STRNOD* NAME      0
3      A      EQU      $NODE
4      K      DO      $SL(F(1)) ,A(K) EQU $SS(F(1),K)
5      END      A
```

The value of STRNOD('ABC') looks like the node produced by the expression \$L1('A','B','C').

### Example 3

STRNOD has an inverse function that converts a node reference, whose selections are string-valued into a string and can be written as follows:

```
1      F      FUNC
2      NODSTR* NAME      0
3      A      EQU      ','
4      K      DO F(1) ,A EQU A:F(1,K)
5      END      A
```

The value of NODSTR(\$L1('A','B','C')) is 'ABC'.

### \$FUNC — Define a Function

When MASM interprets the function body, it assigns the parameter tree value to the label on the \$FUNC directive. If this label is externalized, the label is defined as a function name with no entry parameter, whose entry point is at the beginning of the function body.

## 6.7.3. Nesting of Procedures

Procedures can be nested statically and dynamically. Static nesting of procedures occurs if the lines composing the body of one procedure are physically included in the body of another outer procedure. Dynamic nesting occurs when one procedure calls on another.

A procedure that is statically nested can be referenced only from the procedure containing it initially, although an outer procedure can externalize an entry point to an inner procedure. The procedure sample for an inner procedure is saved when the outer procedure is called and is discarded when the outer procedure is terminated, unless some reference to the inner procedure sample still exists. For efficiency, static nesting should be avoided.

Dynamic nesting of procedures (and functions) produces new levels of the dictionary. Each new procedure (or function) call begins a subassembly and creates a new principal level of the dictionary for symbol lookup and insertion. Procedures written at the same static physical level can be nested at greatly differing dynamic levels. The symbols at more shallow levels in the dictionary are available for definition and reference on each subassembly. A local level of the dictionary exists for symbols whose existence is limited to the current subassembly.

An external symbol can be referenced as an operand if there is no local label with the same name; this includes the parameter node of higher level procedures if all the procedures have distinct \$PROC directive line labels. An external symbol can be used in the label field of a line only by affixing the proper number of asterisks to it, and before any selector group. Each asterisk indicates one level of externalization. Externalization can also create new external symbols at a higher level in the same way.

The number of asterisks used must be the same for each label field reference to ensure that the same variable is obtained. Waiting labels can be externalized also; one of the asterisks is ignored in counting levels of externalization, since the first asterisk indicates waiting label definition. Level counting begins at the level where the waiting label exists, which is already an external level.

### 6.7.4. Waiting Labels

When a label is written in the label field of a procedure call, a definition is not immediately established for it, except for a words-given procedure. Instead, MASM makes a summary pass over the procedure body. If a line with an asterisk in column 1 is encountered during the summary pass, MASM defines the label for the procedure call line at its proper dictionary level external to the procedure, by assigning it the value it should be given if it appeared in the label field of the line with the asterisk in column 1. If MASM finds no such line, it gives the label on the procedure call line the default value of the address of the first word generated by the procedure. Otherwise, if the procedure generates no words, MASM assigns the current location counter value at the start of the procedure subassembly. This type of label within the procedure is called a waiting label, and it can be examined with the \$LF built-in function.

**Note:** *The \$NIL directive can prevent MASM from assigning the default or any other definition to a waiting label.*

### 6.7.5. \$LF(e) — Label Field Description

The value of \$LF is a description of the waiting label for subassembly  $e$ , where  $e$  is a positive integer. For purposes of this function, the active subassemblies are assumed to be numbered from 0, starting with the current subassembly. A T flag is produced if the subassembly does not exist. A V flag is produced if the subassembly is protected. The value of \$LF is a new node reference. If the  $e^{\text{th}}$  subassembly has no waiting label, the node has no selectors defined. If there is a waiting label, the zero selector points to a string in the system character set that represents the identifier. If the waiting label is a selector definition (that is, a subscripted label), then selector  $i$  is the integer value of the  $i^{\text{th}}$  subscript.

\$LF(0) is always illegal, since the current subassembly is the line containing the \$LF call and is always protected. The argument of 1 for \$LF retrieves the waiting label for the procedure containing the line on which \$LF is called.

#### Example 1

The value of \$LF( $e$ )=0 is 1 if there is no waiting label. The value of \$LF( $e$ )=1 is 1 if the waiting label is a simple identifier. The value of \$LF( $e$ )>1 is 1 if the waiting label is a selector definition.

#### Example 2

If the current subassembly has a waiting label of CLB(4,3), then the value of \$LF(1) is \$L0('CLB',4,3).

#### Example 3

The following function converts the output of \$LF to a string corresponding to the label:

```

F      FUNC
CLF*   NAME
      IF      F(1)>1
A      EQU    '(':$CD(F(1,1))
K      DO     2,F(1)-1 ,A EQU A:',':$CD(F(1,K))
A      EQU    A:')'
      ELSE
A      EQU    ''
      ENDF    . THE NULL STRING
      END     F(1,0):A

```

This function can then be used, in the following procedure

```
INCR*      PROC      *0
*          EQU        [CLF($LF(1))]+1
          END
```

that increments the label in the label field of the procedure call line, as follows:

```
K          INCR
```

This function has the same effect as the following line:

```
K          EQU        K+1
```

### 6.7.6. Location Counter Control in Procedures

Any procedure can specify an initial location counter to be used for its generated data by coding the third field of the \$PROC directive. If that field is void, the location counter in effect when the procedure was called is used as the initial location counter. The number of the initial location counter can be retrieved by the \$ILCN function. The location counter in use is reset to the initial location counter at the start of each pass, so a two-pass procedure need not take special action if a permanent location counter change is coded within the procedure. After completing all passes of a procedure, the location counter in use is reset to the one in use when the procedure was called, if it is different from the initial location counter and no permanent location counter change is made. A procedure cannot change the location counter in use, unless specified on the \$PROC directive line. Be careful when changing location counters in procedures because the effects of the change can carry through higher assembly levels.

### 6.7.7. Use of \$NAME and \$GO Directives

In addition to providing an entry point for a procedure, the \$NAME directive can also create a target for the \$GO directive. In the main assembly, a \$GO directive can transfer control only forward to a \$NAME label not yet encountered. In a procedure, a \$GO directive can transfer control forward, backward, or outside the procedure.

If used only as a \$GO target from within the same procedure, the label on a \$NAME line need not be externalized, since a forward \$GO to an unknown name searches to the end of the procedure, while a backward \$GO refers to a name already passed and therefore known. (One exception is if the name is defined by a \$NAME line that appears earlier in the procedure body than the point where the procedure is entered; in this case, a backwards \$GO to a nonexternalized name is unsuccessful.)

If a \$GO directive transfers outside the procedure to another procedure, a diagnostic G flag is generated. Such a \$GO directive is a lateral transfer and does not introduce a new level of definition for the dictionary. The environment in the destination procedure, including the parameter tree, is the same as before the \$GO directive was interpreted. Thus, a lateral transfer can use common code without a procedure call and can save the time required to create and later delete a new dictionary level.

A \$NAME line label, if external, can be used as an entry to the procedure containing it. Only if entry is made by a \$NAME label (internal name) can the zero parameter list be referenced. You can also use a set of internal names to allow one procedure to perform a number of distinct but related actions through the different locations of its entry points. Interpretation of a procedure body begins with the line following the entry point.

### 6.7.8. Types of Procedures

There are three types of procedures in MASM, depending on whether the first subfield is flagged or whether the second subfield exists. An existing second subfield overrides a flagged first subfield. The types of procedures are as follows:

- Two-pass (no flag, void second subfield)
- One-pass (flagged first subfield, void second subfield)
- Words-given (existing second subfield)

Their characteristics are summarized briefly in Table 6-1, where the passes referred to in the heading are the passes made by the next higher subassembly.

**Table 6-1. Characteristics of Procedure Types**

Type of Procedure	Action Taken on Summary Pass	Action Taken on Generative Pass	Restrictions
Two-pass (no-flag, 2nd subfield not given)	Summary pass	Summary and generative passes	None
One-pass (1st subfield flagged)	Summary pass	Generative pass	No forward references
Words-given (no flag, 2nd subfield given)	Increment location counter by number of words given	Generative pass	No forward references, generate number of words specified, no change to location counter

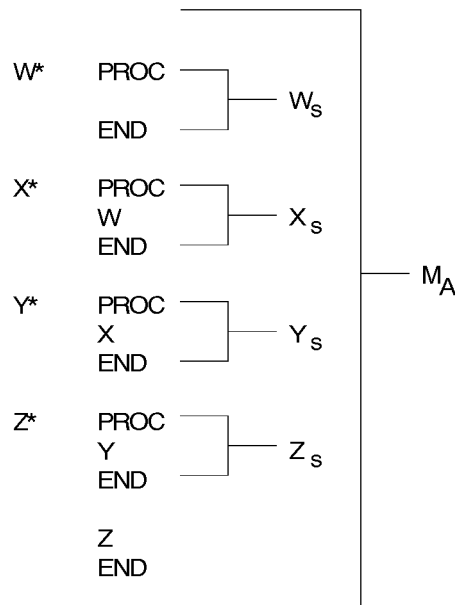
The number of passes performed on a procedure depends on the kind of pass being performed by the next outer subassembly and on the type of procedure being used. This leads to wide variance in efficiency among types of procedures.

On the procedure call line itself, if the number of fields specified is less than the number permitted by the first \$PROC directive subfield value, the procedure call must terminate with a period-space. This avoids scanning a comment as possible parameters for the procedure.

### 6.7.8.1. Two-Pass Procedures

Two-pass procedures have no restrictions. All operations permitted in the main assembly are permitted in a two-pass procedure. In addition, you can use the \$GO directive to transfer backward or into another procedure, as well as forward. (In the main assembly, a \$GO directive can go only forward, not backward or into a procedure.) The number of words generated by distinct calls on a two-pass procedure need not be the same. You can use forward references to labels local to the procedure and manipulate variables external to the procedure. The flexibility achieved can require substantial processing by MASM. Since a two-pass procedure requires two passes during the higher level generative pass as well as the summary pass in the higher level summary pass, the number of passes made by the innermost procedure in a nest of calls to two-level procedures is an exponential function of the depth of nesting. This can be extremely expensive. Consequently, two-pass procedures should be converted to one-pass or words-given procedures where possible.

The following example is a series of nested two-pass procedures.  $W_s$ ,  $X_s$ ,  $Y_s$ , and  $Z_s$  are procedure subassemblies.  $M_A$  is the main assembly. Figure 6-1 indicates the number of passes performed and the values of the built-in functions in the example.





M	Z	Y	X	W	\$FP	\$LP	\$GP
S	— S	— S	— S	— S	1	0	0
G	— S	— S	— S	— S	1	1	0
	G	— S	— S	— S	1	1	0
		G	— S	— S	1	1	0
			G	— S	0	1	0
				G	1	1	1

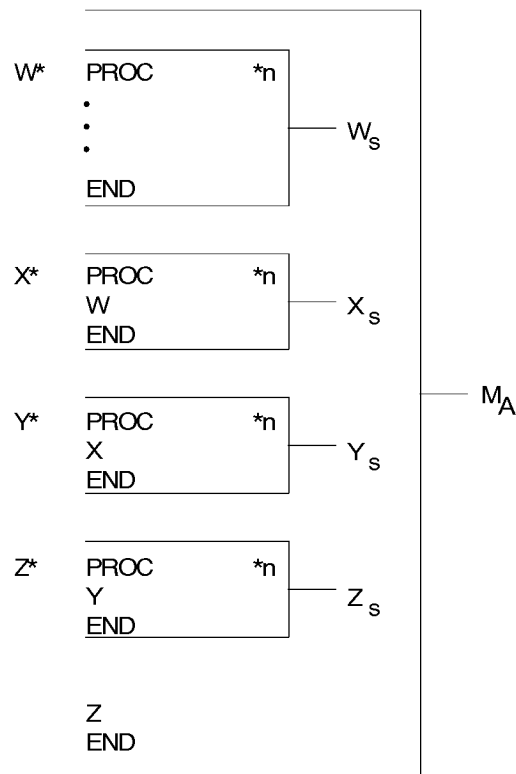
Figure 6-1. Two-Pass Summary

S indicates a summary pass being performed and G indicates a generative pass being performed. The values associated with the built-in functions \$FP, \$LP, and \$GP are as if the functions were in procedure W.

### 6.7.8.2. One-Pass Procedures

One-pass procedures, as their name implies, require only one pass during the generative pass of the next higher level subassembly. Since the omitted pass is a summary pass, the definitions of labels local to the subassembly are not available until after they occur. This is why forward references are not allowed. By eliminating one summary pass, the number of passes made for the innermost nested procedure call does not grow exponentially with the nesting depth. One-pass procedures also avoid the problem of double alteration of external variables and do not need the \$FP function.

The following example is a series of nested one-pass procedures.  $W_s$ ,  $X_s$ ,  $Y_s$ , and  $Z_s$  are procedure subassemblies.  $M^A$  is the main assembly. Figure 6-2 indicates the number of passes performed and the value of the built-in function in the example.



M		Z		Y		X		W		\$FP		\$LP		\$GP
S	—	S	—	S	—	S	—	S		1		0		0
G	—	G	—	G	—	G	—	G		1		1		1

Figure 6-2. One-Pass Summary

S indicates a summary pass being performed and G indicates a generative pass being performed. The values associated with the functions \$FP, \$LP, and \$GP are as if the functions were in procedure W.

6.7.8.3. Words-Given Procedures

As their name indicates, words-given procedures must generate the same number of words on all calls. The number of words must be the value computed by the expression in subfield 2 of the \$PROC directive. Words-given procedures need not be scanned at all during the summary pass of the next higher subassembly, since the primary reason for such a scan is to compute the number of words generated by the procedure call, which is already known. As for one-pass procedures, no summary pass is made for a words-given procedure during the generative pass of the next higher subassembly, so forward references are not permitted in a words-given procedure.

Because the body of a words-given procedure is not scanned during the summary pass of the next higher subassembly, there are some additional restrictions whose violation would result in incorrect values for location counter sizes and dictionary values. Therefore, a words-given procedure cannot have a change of location counter, any externalized definitions (other than entries to the procedure), or a definition of a waiting label. If all of these restrictions can be met, the words-given procedure type should be used, as it is the form of procedure which is least expensive in terms of assembly time.

The following example is a series of nested words-given procedures. W<sub>s</sub>, X<sub>s</sub>, Y<sub>s</sub>, and Z<sub>s</sub> are procedure subassemblies. M<sup>A</sup> is the main assembly. Figure 6-3 indicates the number of passes performed and the values of the built-in functions in the example.

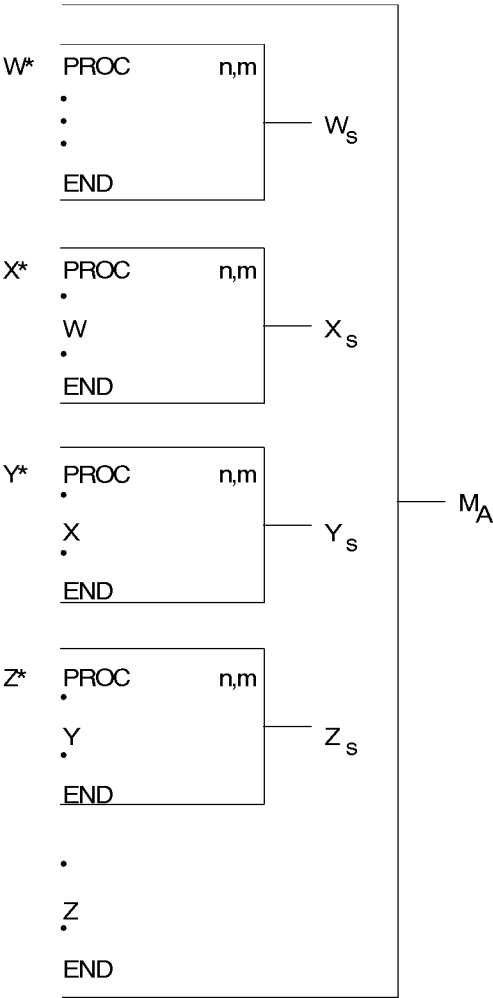




Figure 6-3. Words-Given Procedure Summary

S indicates a summary pass being performed and G indicates a generative pass being performed. The values associated with the functions \$FP, \$LP, and \$GP are as if they are in procedure W.

### 6.7.9. Speeding Up a Two-Pass Procedure

If a two-pass procedure can determine how many words it generates during the first summary pass, it is possible to do only the global operations and skip the interpretation of the generative directives during the first summary pass, instead reserving the proper number of words. This can provide a considerable increase in speed. Similar techniques can be applicable for a one-pass procedure. The method requires an understanding of the \$FP, \$GP, and \$LP built-in functions.

#### Example

```
1      P1*      PROC
2          $IF      (/$LP)**$FP
3          $RES      P1(1,1)
4          $ELSF      $LP**$FP
5      K          $DO      P1(1,1),;
6          +          P1(1,1,K)
7          $ENDF
8          $END
9      B          $EQU      $L1(1,2,3,4,5)
10     C          $EQU      $L1(10,12,14)
11     P1          B
12     P1          C
13     $END
```

#### Explanation

Procedure P1 consists of lines 1 through 8. Line 9 defines a node B with five selectors, which reference the values 1, 2, 3, 4, and 5. Line 10 defines a node C with 3 selectors, which reference the values 10, 12, and 14. Line 11 is a call to procedure P1. When both the main assembly and the subassembly are in the summary pass, the expression  $(/$LP)**$FP$  from line 2 is true, causing line 3 to be interpreted. As a result of line 3, the location counter is incremented by 5. Line 4 causes all images up to line 7 to be skipped. Line 8 indicates the end of the procedure.

When the main assembly is in the generative pass and the procedure is in the summary pass, the expressions  $(/$LP)**$FP$  in line 2 and  $$LP**$FP$  in line 4 are both false. Therefore, statements 3, 5, and 6 are skipped.

When both the main assembly and the procedure are in the generative pass, the expression at line 2 is false. The statement at line 3 is skipped, and the expression in line 4 is true. Therefore, the statements at lines 5 and 6 are interpreted and the result is output.

Line 12 is another call to procedure P1. The same process is repeated with the appropriate location counter increment and the number and value of the words output.

### 6.7.9.1. \$FP — Final Pass

The value of \$FP, which requires no parameters, is 1 if the current subassembly pass is the final pass of the current subassembly. Otherwise, the value is 0. This function should be used to control actions that are performed only once during a subassembly, even though the subassembly can require more than one pass. See also 6.7.9.2, 6.7.9.3, and 6.7.9.4.

### 6.7.9.2. \$GP — Generative Pass

The value of \$GP, which requires no parameters, is 1 if the current subassembly pass is generative. Otherwise, the value is 0. This function should be used to control actions associated with the output, such as listing control and printed displays. See also 6.7.9.1, 6.7.9.3, and 6.7.9.4.

### 6.7.9.3. \$LP — Last Pass

The \$LP function requires no parameters. Its value is 1 if the generative pass of the main assembly is being performed. Otherwise, the value is 0. This function is used to control actions that are performed only after the first (summary) pass of the main assembly is complete. See also 6.7.9.1, 6.7.9.2, and 6.7.9.4.

### 6.7.9.4. Using the \$GP, \$FP, and \$LP Functions

The \$GP, \$FP, and \$LP built-in functions are usually of value only within a procedure. They allow conditional interpretation of code inside a procedure and can speed up the operation of complex procedures.

The \$GP built-in function suppresses calculations that lead only to values for display or data generation until the generative pass is being performed. Computations essential to determining the value of an external symbol or the amount of incrementation of a location counter cannot be postponed to a generative pass, since these determinations are the reason for performing a summary pass.

The \$FP built-in function is used most often to ensure that certain computations are not performed twice. It is meaningful only in two-pass procedures, since no other procedure performs more than one pass per higher level subassembly pass. If an external variable is being incremented in a two-pass procedure, it is incremented on both passes unless protected by the \$FP function.

The \$LP built-in function is useful for both one and two-pass procedures, but not for words-given procedures. \$LP can be used to do reserve operations during the main assembly summary pass and to do data generation based on later defined symbols during the main assembly generative pass. This assumes that the number of words generated on the second pass is the same as the incrementation of the location counter on the first pass.

The applications of these functions is summarized in Table 6-2.

**Table 6-2. Procedure Types Using Pass-Determination Functions**

Procedure Type	Useful Functions
Two-pass	\$LP, \$FP, \$GP
One-pass	\$LP, \$GP
Words-given	None (only one pass made)

### 6.7.10. \$FN(e1,e2) — Form a Name

The value of  $\$FN(e_1,e_2)$  is a procedure or function name. The  $e_1$  parameter must be a previously defined procedure or function name, and  $e_2$  is converted according to the rules for parameter conversion, except that a void expression is also allowed. If  $e_2$  is omitted, a void expression is assumed. The result of this function is a new entry point to the procedure or function named by  $e_1$ ; however, the entry parameter is the value of  $e_2$ . If  $e_2$  is void, the new entry point acts like a procedure or function label, while an existing  $e_2$  produces a new entry point of the name type (the zero selector is defined). The new entry point is assumed to be at the first of the procedure or function body.

#### Example

If PVT is a procedure which has no NAME entry point, the value of  $\$FN(PVT,6)$  is the same as that of the label on the following line,

```
label*          NAME          6
```

if the line is the first line of the procedure.

### 6.7.11. Pass Initialization

MASM initializes to Fieldata at the start of each pass. On subassemblies, MASM initializes to the mode in use when the procedure was called, at each pass through the procedure.



## 6.8. Microstrings

Microstrings are a way of computing the line to be interpreted by MASM by substituting string expressions as parts of a line. Any portion of a line can be generated by microstring substitution with the exception of the line leveler (see 6.9). A microstring is started by a left bracket ( [ ) and is terminated by a right bracket ( ] ). Brackets appearing between single quotation marks as part of a character string are not recognized as beginning a microstring. However, this effect can be obtained by using the concatenation operator. The expression between the brackets must be convertible to a string value in a system character set. The characters of the microstring expression value replace the bracketed expression in the line that MASM is to assemble. All microstring substitution operations take place before any normal interpretation functions, such as label definition, directive recognition, and so on. This means that the directive itself can be computed partially or entirely by microstring substitution.

For example, the following set of definitions

```
1.      A0      EQU      12
2.      A1      EQU      13
      .
      .
      .
16.     A15     EQU      27
```

can be effected by the following line, since the value of \$CD(K) is progressively '0', '1', '2', and so on.

```
K      DO 0,15 ,A[$CD(K)] EQU K+12
```

A less trivial example is provided by a procedure to create conditional jump procedures, as follows:

```
1.      F*      FUNC
2.      END      [P(F(1),*F(2))->'*!'']P(F(1),F(2))
3.      P      PROC      2,2
4.      JE*     NAME      '*TNE'
5.      JNE*    NAME      '*TE'
6.      JEZ*    NAME      'TNZ'
7.      JNEZ*   NAME      'TZ'
8.      K      EQU      P(0,*0)
9.      [P(0,0)] [K->'P(1,1),'!''']F(1,1+K),;
10.      F(1,2+K),P(0,1)+P(1,3+K)
11.      J      F(2,1),F(2,2)
12.      END
```

The preceding procedure creates a number of NAMEs callable by the programmer with two operand fields, the first representing the comparison field (where an A register is required for some tests but not others) and the second field being the jump destination.

## Assembly-Time Controls

---

The following examples show how the preceding procedure can be used:

```
1.      JNE,S3      A0,TABLE,*X6 0,X11
2.      JEZ        FIELD,,H2  *RETURN
```

Line 1 generates the following instructions:

```
TE      A0,TABLE,*X6,S3
J        0,X11
```

Line 2 generates the following instructions:

```
TNZ     FIELD,,H2
J        *RETURN
```

Note in this example, three separate microstrings are used. In the function, a microstring computes either the string \* or the void string, and the result is used as an operator, making the value returned by the function flagged or not. This function also illustrates access from one subassembly (the function) to variables defined in a higher level subassembly (P, which is the parameter tree of the procedure P that called the function F). In the procedure itself, microstrings are used to compute the directive of the test instruction based on which entry point to the procedure was used, and also, whether an a-field is needed by the instruction, and the value for the a-field. The F function is used by the procedure to set up the U and X fields, either of which can have a flag (for indirect addressing or index incrementation).

## 6.9. Levelers

Levelers are perhaps the most difficult to understand of all the MASM concepts. This is partly because, unlike all previous discussions of levels, which pertained to dynamic call nesting levels, levelers pertain to the static nesting levels of procedures, functions, and repeat groups. Levelers apply at the time the sample is saved rather than when it is called for interpretation.

MASM lines and microstrings both can have levelers. A leveler has the following form

```
%n:
```

where  $n$  is an unsigned integer. If the leveler for any line or microstring is omitted, an implicit leveler of %0: is used. The leveler for a line is written preceding the label field; the leveler for a microstring is written immediately after the left bracket.

The following is an example of levelers:

```
%1:ABC      EQU      14
              +      [%2:$CD(I)]
```

The static level of a line is determined by the following algorithm:

- The level at the start of the main assembly is 0.
- The level is incremented by 1 for each \$PROC, \$FUNC, or \$REPEAT directive encountered.
- The level is decremented by 1 for each \$END or \$ENDR encountered.

Thus, when saving a sample, levels are greater than zero. A line is interpreted if the leveler for the line is the same as the level of the line. A microstring is substituted if the leveler for the line plus the leveler for the microstring is equal to the level of the line. Therefore, if the present level is 2 (as it would be inside a function that is inside a procedure), the first line and the microstring on the second line would both be interpreted as follows:

```
%2:A      EQU      'P(1,1)'
%1:      [%1:DIR]
```

It is important to understand that when the static nesting level is being calculated by the programmer, the same line can be encountered by MASM at different times with different levels. For example, if F is a function nested statically within the procedure P, when the sample for P is picked up, lines inside F are at level 2. When P is called, the sample for F as a function (rather than as part of P) is saved, and these lines are at level 1, since the procedure is being interpreted. This makes F part of the main assembly.

## Assembly-Time Controls

---

The following examples clarify this. The first example is a procedure (perhaps a debugging procedure) that generates code only if a global assembly variable is set. The use of levelers allows the code inside the procedure to be deleted from the sample so that any calls to the procedure are faster than if the code were skipped each time the call was performed. In this and the following examples, the level of the line is indicated to the left of the line's label field.

```
0          P          PROC          1,2*DEBUG
1          SNOOP*     NAME          0
1          %1:        F            DEBUG
1          SLJ        SNOOPY$
1          +          P(1,2),P(1,1)
1          %1:        ENDF
1          END
```

The procedure produces the same results without the levelers, but the IF-ENDF group is skipped each time the procedure is called if DEBUG=0. The use of levelers thus saves time in this case. DEBUG must be either 1 or 0 and cannot be changed in the assembly. A much more complicated example is taken from Church's lambda calculus. In this case, a procedure is defined that, when called, defines its waiting label as a function with a given set of arguments. The function computes and returns the value of an expression using those arguments. The dummy arguments and the expression are specified as parameters to the procedure.

```
0          P          PROC          *2
1          LAMBDA*    NAME          0
1          F$         FUNC
2          *          NAME
2          %1:A       REPEAT        P(1)
3          [%1:P(1,A)] EQU          F$([%1:$CD(A)])
3          ENDR
2          END        [%1:P(2,1)]
1          END
0          ...
```

When the sample for this procedure is picked up, no levelers match the level of their line, so no lines are interpreted. Now assume that LAMBDA is called as follows:

```
0          ADD        LAMBDA        'X','Y' 'X+Y'
```

Then the procedure P is interpreted, resulting in the following:

```

0          F$          FUNC
1          *          NAME
1          %1:A        REPEAT      P(1)
2          [%1:P(1,A)] EQU          F$([%1:$CD(A)])
2          ENDR
1          END          [%1:P(2,1)]
0          END

```

Now, several line levels match levelers. Therefore, MASM, as it now scans the body of procedure P, attaches the waiting label to the function F\$ NAME line. REPEAT is then interpreted, since its leveler matches its level. This means that MASM generates the following line P(1) times under control of REPEAT, having first picked it up as the REPEAT sample:

```

1          [%1:P(1,A)]      EQU          F$([%1:$CD(A)])

```

Since the level of this line is matched by the sum of the line's leveler and the leveler of each of the microstrings, MASM performs the micro substitution. For the particular LAMBDA call being considered, MASM thus places the following lines into the function body as part of the process of picking up the function sample. This completes the action of REPEAT.

```

          X          EQU          F$(1)
          Y          EQU          F$(2)

```

Finally, the following line is reached:

```

1          END          [%1:P(2,1)]

```

Again, the microstring leveler plus the (implicit) line leveler matches the level of the line, so the substitution is made, resulting in the function body being completed with the following line:

```

          END          X+Y

```

This completes the work of the procedure. ADD is now defined as a function that computes the sum of its arguments, just as if it had been written as follows (since this is exactly what has been stored as the function body for ADD):

```

F$          FUNC
ADD*        NAME
X          EQU          F$(1)
Y          EQU          F$(2)
          END          X+Y

```

## Assembly-Time Controls

---

You can then code the following statement and MASM will generate the constant 5:

```
+          ADD(2,3)
```

The LAMBDA procedure can be used in a more general way to build up the rest of the lambda calculus in the fashion of LISP. For example

```
FACTORL      LAMBDA      'N' 'N->N*FACTORL(N-1)!1'  
+            +            FACTORL(4)      . 24 GENERATED
```

or at a higher level

```
APPLY        LAMBDA      'F','X' 'F(X)'  
+            +            APPLY(/$,2)      . $(2) GENERATED  
+            +            APPLY(/FACTORL,5) . 120 GENERATED
```

Unlike LISP, MASM requires functions to be identified by the control information operator (/).

A more practical use for levelers than LAMBDA might be the inclusion or deletion of debugging displays from a procedure so that skipping would not have to be done for each call on the procedure. The following example always inserts the lines specified on the DINSERT call, but displays them only if DEBUG is set at the time the procedure is defined:

```
      P          PROC          *1  
DINSERT*  NAME  
K          REPEAT          P(1)  
%2:       IF          DEBUG  
          DISPLAY      P(1,K)  
%2:       ENDF  
          INSERT      P(1,K)  
          ENDR  
          END
```

The following are exceptions to the rules:

- The \$PROC, \$FUNC, and \$REPEAT directives are at the level below the sample that follows them.
- A leveler is not needed on \$END or \$ENDR lines, since they are assumed to match up with the associated \$PROC, \$FUNC, or \$REPEAT directives and thus have an implied leveler that is the same as the associated start of the sample directive.
- The expression in the operand field of a \$NAME line is evaluated when the sample is picked up, so that a leveler of 1 is implied for the expression itself in all cases.

# Section 7

## Dictionary Control

To use MASM effectively, you need a general knowledge of the storage mechanism known as the dictionary. The most elementary function of the dictionary is to retain knowledge of labels and values associated with the labels, such as the value and number of the location counter at the time the label is defined. At processor initialization, the dictionary contains the directives and functions built into MASM.

A name and its value are automatically entered into the dictionary when MASM detects a label on an assembler statement. The value associated with the label is determined by its use in the label field and the rest of the assembler statement. For example, the value associated with built-in directives and functions is not really a value in the normal sense of the word; rather it is control information that acts as data to govern some stage of manipulation, not as data to be manipulated.

Each value entered into the dictionary has a type associated with it. See 3.4 for definitions of the types available.

The instruction mnemonics and MASM directives without a leading “\$” can be redefined even though they have already been initialized in the MASM dictionary. MASM functions and the remaining MASM directives cannot be redefined.

**Note:** *Even though this capability exists, it is not recommended that the instruction mnemonics and MASM directives be redefined. Use of the M option, which allows you to redefine all instruction mnemonics and MASM directives without a leading “\$,” increases assembly time to perform the additional searching. Also, other products or applications that can use the same library files and the M option can be adversely affected by these redefinitions.*

### 7.1. Structure of the Dictionary

The dictionary is structured by levels. These levels define the scope of labels and have the range 0 to  $n$ , with 0 as the highest level and  $n$  as the lowest level. Labels defined at level 0 are known outside the program. Labels defined at level 1 are known only to the program. Labels defined at levels lower than level 1 are known to selected portions of the program.

All operation mnemonics and built-in directives and functions are known at level 1.

The dynamic nesting of subassemblies causes lower levels to be employed. For each subassembly (including the main assembly), there is a principal level of definition in the dictionary. Each new subassembly in a nest introduces a deeper level of definition in the dictionary as its principal level (that is, the value of the principal dictionary level is 1 on the main assembly and is incremented by 1 for each nested subassembly).

Labels defined at a particular level stay at that level unless it is specifically requested that they be known at some higher level.

When a symbol is presented, a value is retrieved. Normal retrieval is accomplished by starting at the level corresponding to the current subassembly and searching progressively higher levels (that is, lower-numbered levels) until the symbol is found.



## 7.2. \$DELETE — Delete a Definition

Call the \$DELETE directive as follows:

```
label      $DELETE
```

There are no parameters, but the label field is required. This directive deletes the final relationship of a definition, which can delete both data and control information, since an identifier can reference both. If the *label* is a selection  $a(s_1, \dots, s_n)$ , the effect of \$DELETE is to delete the selector  $s_n$  of the selection  $a(s_1, \dots, s_{n-1})$ . This means that the \$DELETE directive can be used to eliminate nodes and selectors within tree structures. Nodes and procedure sample blocks, that can be referenced from more than one place, are deleted when all references to them are deleted. The \$DELETE directive can be used in definition mode to remove procedure sample blocks when the procedure is called solely to establish definitions.

### 7.3. \$HASH(e) — Dictionary Class Index

The \$HASH function requires one parameter. The value of the expression *e* should be a string in the system character set.

The value of \$HASH(*e*) is the dictionary class index of an identifier, that is, the system hash value of an identifier.

**Example**

```
1. @MASM,S MASM*MSM4.HA001,TPF$.
2. MASM xxRyy
3
4
5. 0 000000 0000000000021
6. 000001 0000000000102
7. 000002 0000000000102
8
9.
10. LOCATION COUNTERS: $(0) 000003 $(1) 000000
11. END MASM - LINES: 12 TIME: 1.687 STORAGE: 28918
```

1.	\$(0)		
2.	tag	\$equ	'MASM'
3.		+	\$hash('tag')
4.		+	\$hash(tag)
5.		+	\$hash('MASM')
6.		\$end	

**Explanation**

**Line 5**

The class number for the symbol 'tag' is 021.

**Line 6**

The class number for the value of the symbol tag is 0102.

**Line 7**

The class number for the symbol 'MASM' is 0102.

## 7.4. \$IC(*e*) — Identifier Class

The \$IC function takes one argument that must be a binary value, without relocation in the range 0 to 127. The function returns a portion of the MASM symbol dictionary containing all those identifiers whose MASM system hash code is *e* and whose mode matches the current mode indicated by the \$BASIC and \$EXTEND directives. The value of \$IC(*e*) is a new node reference. The selector *m* is defined for this node if there are class *e* identifiers at level *m*. Levels refer to the hierarchy of definitions established by the nest of subassemblies active at the time the \$IC function is called. For each selector *m*, the value of the selection is a reference to a node that is distinct from all previously allocated nodes. The selectors defined for this node are consecutive integers (starting from 1) that select strings in the system character set corresponding to the identifiers defined at level *m*. If the identifier defines a value or node reference, then the string is not flagged. If the identifier defines control information, the string is flagged.

The \$HASH function (see 7.3) computes the class index of an identifier, that is, the system hash value of an identifier.

### Example 1

```
$IC($HASH('MASM'))(1,5)
```

This is a string that is the name of the fifth identifier in the class defined by the value of the expression \$HASH('MASM').

Level 0 in the dictionary is the level of symbols defined outside the main assembly (external symbols for generative assemblies, saved symbols for DEF mode assemblies). Level 1 is the level of the main assembly itself. Higher numbered levels are those of progressively deeper nested subassemblies (procedures and functions).

### Example 2

```
$IC(14)(1,5)
```

This is a string that is the name of the fifth identifier in class 14 defined at the level of the main assembly. This string can be used in a microstring expression to retrieve the identifier itself, so a symbol table can be constructed for use at execution time. Because the dictionary is dynamically re-ordered, \$IC(14)(1,5) is not necessarily the same identifier each time the function is called.

### 7.5. \$LEVEL — Dictionary Level Control

The format of the \$LEVEL directive is as follows:

```
$LEVEL       $e_1, e_2, e_3$ 
```

where  $e_1$ ,  $e_2$ , and  $e_3$  are binary values without relocation. For each subassembly (including the main assembly), there is a principal level of definition in the MASM symbol dictionary. Each new subassembly in a nest introduces a deeper level of definition in the dictionary as its principal level.

The value of  $e_1$  on the \$LEVEL directive establishes the dictionary level for the following lines (up to the next \$LEVEL or the end of the subassembly) as being  $e_1$  levels deeper than the current principal level. The value of  $e_2$  determines the dictionary insertion level for new symbols.  $e_2$  is always positive and indicates that symbols must be defined at more shallow levels, just as if they had been written in the label field with trailing asterisks. The value of  $e_3$  determines the level at which the dictionary search for identifiers begins. Identifiers defined deeper than this level are not found.

For the main assembly, the level more shallow than its initial principal level is that of symbols external to the assembly itself. These symbols are either the external definitions in the preamble of the relocatable element or, for definition mode assembly, the symbols retained in the dictionary snapshot written out as the omnibus element. Therefore, for an ordinary assembly, the following example externally defines all following symbols defined at the current level:

```
LEVEL      0,1,0
```

For a definition mode assembly, this example retains all following symbols in the dictionary snapshot in the output element. Using this form eliminates the need for externalizing explicitly (with an asterisk) all of the symbols defined by the element.

## 7.6. \$LEV — Principal Dictionary Level

The format of the \$LEV function is as follows:

\$LEV

The \$LEV function requires no parameters. Its value is the number of the principal dictionary level. The value of the principal dictionary level is 1 on the main assembly and is incremented by 1 for each nested subassembly.

### Example

1.	P*	PROC
2		+\$LEV
3		END
4		+\$LEV
5		P
6		END

### Explanation

Line 4

The value of \$LEV is 1.

Line 5

When procedure P is called, the value of \$LEV at line 2 is 2.

## 7.7. \$XLEV — Current Dictionary Level

The format of the \$XLEV function is as follows:

\$XLEV

The \$XLEV function requires no parameters. The value returned by \$XLEV is the number of levels (or asterisks) needed to raise a label from the current dictionary insertion level to level 0 of the main assembly. This cannot be the same result as acquired by the \$LEV function that returns the principal subassembly level (see 7.6). The difference occurs when a \$LEVEL directive is used to change the dictionary insertion level for new symbols.

### Example

```

1. @MASM,S MASM*MSM4.XL001,TPF$.
2. MASM xxRy
3.
4.
5.
6.
7.
8.
9. L      0000000000001
10.      0000000000000
11. L
12.
13.
14. ENTRY POINTS:      LAB1 0000000000002      LAB2 0000000000001      LAB3 0000000000001
                        LAB4 0000000000000
15. ASSEMBLY CONTAINS ERRORS: - FLAGS: L
16. END MASM - LINES: 29 TIME: 1.737 STORAGE: 28970 ERRORS: L(2)

```

### Explanation

#### Line 3-7

Source lines 1 through 5 are processed when the proc PCALL is called at line 11 (source line 9).

#### Line 8

The dictionary insertion level of labels in this subassembly is raised one level.

#### Line 9

The \$SR (string repeat) function is used inside a microstring to append one asterisk to the label LAB3. The resulting label LAB3\* generates an L flag for a label above level 0. The value of LAB3 is 1, equal to the value of the \$LEV function.

#### Line 10

No asterisks are appended to the label LAB4 because the value of \$XLEV is 0. The entry point LAB4 is generated at level 0 and is equal to 0.

#### Line 11

Call the proc PCALL that generates the labels LAB1 and LAB2.

### Line 5

Two asterisks are appended to LAB1. This makes LAB1 external to the proc PCALL. The \$LEVEL directive in the calling assembly affects this label. An L flag is issued for generating LAB1 above level 0 of the main assembly.

### Line 6

One asterisk is appended to LAB2, which makes it external to the proc PCALL. LAB2 is raised a level by the \$LEVEL directive in the calling assembly. LAB2 is generated at level 0 of the main assembly.

### Line 14

The line that begins with ENTRY POINTS shows the value assigned by \$EQU to the entry points generated by this example.

## 7.8. Example — Value Retrieval

### Example

```

@MASM,S ...
0000000000001      1.      LAB1*      $EQU      1
                    2.      P*        $PROC
                    3.
                    4.      +          LAB1
                    5.      P2*        $PROC
                    6.      LAB1      $EQU      3
                    7.      LAB1*     $EQU      2
                    8.      +          LAB1
                    9.      $END0
000000 0000000000002 10.      +          LAB1
000001 0000000000002 11.      P
000002 0000000000003 12.      P2
                    13.      $END

LOCATION COUNTERS:  $(0) 000003  $(1) 000000
ENTRY POINTS: LAB1 0000000000001
END MASM - LINES: 44 TIME: 0.565 STORAGE: 8013

```

### Explanation

This example is explained by line number, in the order in which MASM processes the lines. The line number in the left hand column corresponds to the line number of the statements in the MASM example.

### Summary Pass — Pass 1 of the main assembly

#### Line 1

The current processing level of this line is 1 (the main assembly). The symbol LAB1 is defined at level 0 (external) of the dictionary because of the asterisk appended to it.

#### Line 2

This line sets up the procedure P. A sample save is performed and P is placed in the MASM dictionary. Lines 2 to 4 are processed as a part of a subassembly when P is called at line 11.

#### Line 5

This line sets up the procedure P2. A sample save of P2 places it in the dictionary. Lines 5 to 9 are processed as a part of a subassembly when P2 is called at line 12.

#### Line 10

Nothing is generated during the summary pass of the assembly.

#### Line 11

This line invokes a subassembly in which P is processed. The nesting level and processing level of the subassembly is 2. Lines 2 to 4 are processed at this time. Remember that two assembly passes are performed, and the value of LAB1 is not generated until the second (generative) assembly pass.



### Line 12

This line invokes a subassembly in which the proc PS is processed. The nesting level and processing level of the subassembly is 2. Lines 5 to 9 are processed at this time. Labels are inserted into the dictionary on the summary pass (pass 1) of the subassembly.

### Line 6

The label LAB1 is inserted into the dictionary at level 2, the current principal dictionary level of the subassembly.

### Line 7

Because of the appended asterisk, this label is known at the level of the calling assembly. When this line is processed, LAB1 is inserted at level 1 of the MASM dictionary. This means that LAB1 is now defined at 3 different levels of the dictionary.

### Line 8

Nothing is generated on the summary pass of the subassembly.

### Line 9

This ends the subassembly invoked by line 12. Level 2 of the dictionary is deleted at this time.

### Line 13

This line ends the summary pass of the main assembly. All levels of the dictionary except level 0 and 1 are deleted.

## **Generative Pass — Pass 2 of the assembly**

### Line 1

LAB1 is defined for the second time at level 0 of the dictionary.

### Line 2

A sample save is performed for procedure P.

### Line 5

A sample save is performed for procedure P2.

### Line 10

A value is retrieved for LAB1. The dictionary is searched, starting at the current principal level (1). The value of LAB1 found at level 1 is generated (000000000002).

### Line 11

This line invokes a subassembly in which P is processed. The current principal dictionary level is 2. A summary and generative pass of P is performed. The summary pass is identical to the summary pass performed on pass 1 of the main

## Dictionary Control

---

assembly. On the generative pass of the procedure, any instructions or values are generated.

Line 3

A value is retrieved for LAB1. The dictionary is searched, starting at the current principal dictionary level (1). The definition for LAB1 at level 1 is found and generated (000000000002). This definition of LAB1 was placed in the dictionary when P2 was processed on the first assembly pass.

Line 9

This ends the subassembly invoked by line 11. Level 2 of the dictionary is deleted at this time.

Line 12

A summary and generative pass of P2 is performed. The summary pass is identical to the summary pass performed on pass 1 of the main assembly.

Line 6

LAB1 is defined at level 2 of the dictionary.

Line 7

LAB1 is defined at level 1 of the dictionary.

Line 8

A value is retrieved for LAB1. The dictionary is searched, starting at the current principal level (2). The definition for LAB1, found at level 2, is generated (000000000003).

Line 9

This ends the subassembly invoked by line 12. Level 2 of the dictionary is deleted at this time.

Line 13

The generative pass of the assembly is finished, and the relocatable element is generated.

# Section 8

## Processor Information Functions

This section describes functions that retrieve the following:

- Date and time information
- The line counter value
- The processor call parameters
- The MASM operating mode settings

### 8.1. \$DATE — Date and Time Function

The format of the \$DATE function is as follows:

\$DATE

The \$DATE function requires no parameters. It returns a 12-character string consisting of the date and time in ER DATE\$ format (*mmdyyhhmmss*). The string that is returned is either Fielddata or ASCII, depending on the character type in effect at the time the \$DATE function is called.

#### Example

1.	AB	\$EQU	\$DATE	
2.		\$DISPLAY	\$SS(AB,1,6)	. Extract date
3.		\$DISPLAY	\$SS(AB,7,6)	. Extract time

#### Explanation

Line 1

Equates AB to a Fielddata character string of the date and time returned by the \$DATE function

Line 2

Extracts the date (in *mmdyy* form) and displays it

Line 3

Extracts the time (in *hhmmss* form) and displays it

### 8.2. \$LINES — Line Counter

The format of the \$LINES function is as follows:

```
$LINES
```

The \$LINES function requires no parameters. It returns a count of the number of lines scanned by MASM since the beginning of the assembly. The lines counted include those from source input, library procedures, procedure and function interpretation, sample scanning, \$REPEAT and \$DO repetitions, \$INSERT images, and images skipped by \$GO and \$IF. Continuation images are considered part of the initial line and are not counted separately. The final value of the line counter is printed at the end of an assembly, if the appropriate type of listing is requested.

The line counter provides a simple measure of the cost of the assembly process, which is adequate for most purposes of optimization. The following lines illustrate how this function can be used to measure the cost of a given section of code:

```
K          EQU          $LINES
              .
              .          code being measured
              .
          DISPLAY      $CD($LINES-K-1):' LINES'
```

### 8.3. \$PAR(e) — Processor Call Parameter

The format of the \$PAR function is as follows:

```
$PAR(e)
```

The \$PAR function provides access to the parameters on the MASM processor call statement. The argument *e* must be a binary value without relocation in the range 0 to 63. If *e* is zero, the value of \$PAR is the option bits from the MASM call in master bit notation (bit 0 corresponds to Z, bit 1 to Y, and so on). For *e*>0, the function returns the element name subfield of field *e* of the MASM processor call statement. If no such field exists, a void string is returned. This function can make assembly actions depend on parameters specified from outside the MASM environment.

## 8.4. \$TMODES — MASM Operating Mode

The format of the \$TMODES function is as follows:

\$TMODES

The \$TMODES function requires no parameters. \$TMODES returns a binary value with bits set or cleared. Table 8-1 shows the meanings for each bit of the returned value.

**Table 8-1. Mode Bit Settings for \$TMODES**

Bit	Setting	Meaning
0*	1	\$ASCII directive in effect
	0	\$FDATA directive in effect
1	1	\$LIST directive in effect
	0	\$UNLIST directive in effect
2	1	\$OCTAL directive in effect
	0	\$HEX directive in effect
3	1	Basic-mode code generation
	0	Non-basic mode
4	1	Extended mode code generation
	0	Non-extended mode
5	1	\$DEF mode assembly
	0	Not a definition mode assembly
6	1	\$SYM mode assembly
	0	Not a \$SYM mode assembly
7	1	\$REL mode assembly
	0	Not a \$REL mode assembly
8	1	\$OBJ mode assembly
	0	Not a \$OBJ mode assembly

continued

\* Least significant bit

**Table 8-1. Mode Bit Settings for \$TMODES (cont.)**

Bit	Setting	Meaning
9	1	\$FORCE directive in effect
	0	Not a forced relocation assembly
10	1	18-bit operands allowed for certain architecture-specific instructions
	0	18-bit operands not allowed for certain architecture-specific instructions

\* Least significant bit

Bits are defined for \$DEF, \$SYM, \$REL, and \$OBJ. With 0 being the least significant (rightmost) bit, the bit assignments are \$DEF mode = 5, \$SYM = 6, \$REL = 7, and \$OBJ = 8. Logical operations can be performed on the \$TMODES function to produce values for conditional expressions. See Table 8-2 for some examples of common expression usage.

**Table 8-2. Expressions Used With \$TMODES**

Expression	Result
\$TMODES**040	\$DEF Mode
\$TMODES**0100	\$SYM Mode
\$TMODES**0200	\$REL Mode
\$TMODES**0400	\$OBJ Mode
\$TMODES**0770=0210	Basic mode relocatable binary generation
\$TMODES**0770=0420	Extended mode object module generation
\$TMODES**0600	Code generation allowed
\\$TMODES**0600	Code generation not allowed

# Section 9

## MASM Output

This section describes relocatable binary output and the use of the \$INFO directive to communicate between MASM and the Collector.

### 9.1. Types of MASM Output

MASM generates the following types of output:

- Relocatable binary (RB) element, using the \$REL directive (see 9.2)
- Object module, using the \$OBJ directive (see Section 10)
- Symbolic output (SO), using the \$SYM directive (see Section 11)
- Definitions, using the \$DEF directive (see Section 12)

You must select the type of MASM output during the summary pass of the main assembly. Do this by placing the appropriate directives at the beginning of the source code. The \$TMODES function (see 8.4) provides the capability to determine which output mode is in effect.

### 9.2. \$REL — Relocatable Binary Output

The \$REL directive requires no parameters. With this directive, MASM uses the Collector library relocatable output routine, ROR\$E, to produce a relocatable integer element. This is the default mode of the assembler.

## 9.3. \$INFO — Special Information

The \$INFO directive is used to communicate between MASM and the Collector. It is called by the instruction

```
label      $INFO      e0 e1,...,en
```

where  $e_0$  is an integer group value without relocation in the range 1 to 12. The meanings of  $e_1, \dots, e_n$  and that of the call itself depend on the value of  $e_0$ , which is referred to as the group number.

### 9.3.1. Processor Mode Settings (Group 1)

The format of \$INFO 1 is as follows:

```
$INFO      1  e1
```

This type of call controls the arithmetic fault mode and quarter-word or third-word sensitivity of the output element. The parameter  $e_1$  is an integer value in the range 0 to 077, which is treated as a bit mask whose bits have the meanings shown in Table 9-1.

**Table 9-1. Bit Meanings for \$INFO Group 1**

Bit	Meaning
0*	Specify quarter-word or third-word sensitivity
1	Quarter-word sensitive
2	Third-word sensitive
3	Specify arithmetic fault mode
4	Arithmetic fault compatibility mode
5	Arithmetic fault noninterrupt mode

\* Least significant bit

If the current subassembly pass is not generative, the directive is ignored. If bit 0 is set, the values of bits 1 and 2 are substituted for bits 25 and 26 of the flag bits word of the element table entry for the relocatable and symbolic output elements. Similarly, if bit 3 is set, the values of bits 4 and 5 are substituted for bits 29 and 30 of the flag bits word.

For example, the output relocatable element can be marked as quarter-word sensitive by setting bits 0 and 1 as shown by the following line:

```
$INFO      1 3
```



### 9.3.2. Common Block (Group 2)

This call specifies that a location counter is a common block. The format of \$INFO 2 is as follows:

```
$INFO 2  $e_1, e_2, [e_3]$ 
```

where:

$e_1$

A string specifying the common block name.

$e_2$

The location counter number used to reference the common block.

$e_3$

The minimum address of the location counter (optional).

The parameters  $e_2$  and  $e_3$  must be integer values without relocation. The string  $e_1$  must satisfy the Collector requirements for a common block name (no embedded blanks, commas, or periods). The effect of this call is to make  $e_2$  refer to the common block named by  $e_1$  with minimum address  $e_3$ . If there are several directives with  $e_1$  and  $e_2$  identical but different values for  $e_3$ , the largest of these values is used.

You must allocate space for the location counter if it is included in the output element preamble. Therefore, a common block location counter must be incremented somewhere in the element referencing it, either by the \$RES directive or by data generation.

For example, the common block COMDATA would be made available as location counter 4 by the following line:

```
$INFO 2 'COMDATA',4
```

### 9.3.3. Minimum D-Bank Specification (Group 3)

The format of \$INFO 3 is as follows:

```
$INFO 3  $e_1$ 
```

This call specifies the minimum address for the D-bank of the element containing the \$INFO 3 directive. The parameter  $e_1$  is a nonnegative, nonrelocatable integer value that specifies the minimum address. This directive is ignored for nongenerative passes. If there are several such \$INFO directives, the largest value specified is used.

### 9.3.4. Blank Common Block (Group 4)

This call specifies that a location counter is a blank common block. The format of \$INFO 4 is as follows:

```
$INFO      4   $e_1, e_2$ 
```

where:

$e_1$

The location counter number used to refer to blank common.

$e_2$

The minimum location counter address (optional).

The interpretations for  $e_1$  and  $e_2$  are the same as for  $e_2$  and  $e_3$ , respectively, for a group number of 2. Blank common can also be referred to by a group number 2 \$INFO directive with a name of 'BLANK\$COMMON' for  $e_1$ .

### 9.3.5. External Reference Definition (Group 5)

The format of \$INFO 5 is as follows:

```
$INFO      5   $e_1$ 
```

This call creates an external reference to a symbol that uses characters not permitted in a MASM identifier. The parameter  $e_1$  is a string that specifies the name of the external identifier. It is limited to 12 left-justified, space-filled characters, from the Fieldata character set. The label in the label field is equated to a value of 0 with full value relocation by the identifier  $e_1$ .

For example, the following line allows a MASM element to reference the external symbol PL\$SCAN by using the label PL\$CALL. PL\$SCAN is defined in another element and written in a different language.

```
PL$CALL      $INFO      5  'PL$SCAN'
```

### 9.3.6. Entry-Point Definition (Group 6)

The format of \$INFO 6 is as follows:

```
$INFO      6   $e_1, e_2$ 
```

This call performs the same operation as group 5, but applies to an entry-point name. It allows creation of an entry point whose external name contains characters not allowed in a MASM identifier. The parameter  $e_1$  is a string (restricted to 12 characters, left-justified, space-filled, from the Fieldata character set) specifying the name of the identifier.  $e_2$  is an integer value that specifies the value of the entry point. If the

identifier specified by the string  $e_1$  is given more than one definition, only the last one is transmitted to the preamble of the relocatable output element.

For example, the following line makes the value of VALR (an internally defined symbol) available to programs written in another language as the external symbol VALREF:

```
$INFO      6 'VAL\REF',VALR
```

### 9.3.7. Even Starting Address (Group 7)

The format of \$INFO 7 is as follows:

```
$INFO      7   $e_1$ 
```

This call specifies that a location counter must be given an even program absolute starting address. The parameter  $e_1$  specifies the number of the location counter and must be an integer and nonrelocatable.

### 9.3.8. Static Diagnostic Information (Group 8)

The format of \$INFO 8 is as follows:

```
$INFO      8   $e_1$ 
```

This call specifies that a location counter is a part of the static diagnostic information that is part of the absolute element diagnostic tables, rather than a part of a segment. The parameter  $e_1$  is a relocatable integer value that specifies the number of the location counter to be used. Data generated under a group 8 location counter are given their correct values, except that the relocation base of a group 8 location counter is always set to 0 by the Collector. Diagnostic routines can reference this information at execution time.

### 9.3.9. Read-Only Location Counters (Group 9)

The format of \$INFO 9 is as follows:

```
$INFO      9   $lc_1, lc_2, \dots, lc_n$ 
```

where  $lc_1, lc_2, \dots, lc_n$  are location counters that are to be marked as read-only. These location counter numbers must be nonrelocatable and integer.

\$INFO 9 indicates that the specified location counters are read-only. The Collector uses this information to mark as read-only any nonsegmented bank containing only read-only (\$INFO 9) location counters.

### 9.3.10. Extended Mode Location Counter (Group 10)

\$INFO group 10 marks the specified location counters as requiring extended mode. The format of \$INFO 10 is as follows:

```
$INFO      10   $e_1, e_2, \dots, e_n$ 
```

where  $e_1$  through  $e_n$  are nonrelocatable integer values in the range 0 to 63. The parameters specify the location counters to be marked as extended mode only.

### 9.3.11. Void Bank (Group 11)

\$INFO group 11 defines a void bank. The format of \$INFO 11 is as follows:

```
$INFO      11   $e_1, e_2, e_3$ 
```

where:

$e_1$

is a string specifying a bank name.

$e_2$

is the starting address of the bank. If not specified, zero is assumed.

$e_3$

are the following option bits:

bit 0	S option bank (shared bank)
bit 1	D option bank (dynamic bank)
bit 2	I-bank (instruction bank)

**Note:** *Bit 0 is the least significant bit.*

Parameters  $e_2$  and  $e_3$  are nonrelocatable integer values. Multiple void banks can be defined and multiple occurrences of the same void bank are allowed.

Void banks are assumed to be D-banks unless the I option bit is set.

### 9.3.12. Library Search File (Group 12)

\$INFO group 12 modifies the library file search order used by the Collector. The format is as follows:

```
$INFO      12  e1
```

where  $e_1$  is a string expression specifying a keyword.

The keyword is defined by a COMUS or SOLAR installation to identify a file of relocatable elements. Keywords are related to a file by use of an integer index. The index is used by the Collector to select the appropriate relocatable library file. The keyword must conform to the rules for OS 1100 Exec names. Refer to the *OS 1100 Collector Programming Reference Manual*.

#### Example

```
INFO      12  'FORTRAN'
```

### 9.3.13. Restrictions

Do not use a location counter with more than one of the group numbers 2, 4, 7, and 8. Only one location counter can be a blank common block or a labeled common block with a given label.

The first six characters of an identifier used for a common block name, an external reference, or an entry point cannot be zero or negative zero when converted to Fielddata.



# Section 10

## Object Modules

Object modules are the functional replacement for both relocatable and absolute elements. The object module environment contains more control information for the operating system than do the relocatable and absolute environments. This gives the programmer more control over how the programs are linked and loaded. Object modules are made up of smaller units called bank groups (see 10.2) and banks (see 10.3). Associated with each bank are attributes that determine its use. These attributes are described in 10.3.1 and their applications are explained in 10.3.2. There are attributes associated with symbolic references (see 10.3.4) and definitions (see 10.5) also. See the *OS 1100 Linking System Programming Reference Manual* for a description of each attribute and its defined values.

To make it easier to specify the many attributes required for an object module, a MASM definition element (see Section 12) is provided. This element contains numeric codes that are used as values for most of the attributes, forms that help build composite values of several attributes, and procedures for generating references via link vectors (see 10.6). The definition element is released with MASM in element OM\$DEF and should be placed in SYS\$LIB\$\*MASM or otherwise made available to the assembly.

**Note:** *Unisys supports “extended mode” MASM usage (assembler directive \$EXTEND with or without assembler directive \$OBJ) only in software written by Unisys or in interfaces written by the customer that explicitly require extended mode assembler-produced elements according to the documentation written by Unisys. In a “nonextended mode” MASM usage (absence of assembler directive \$EXTEND), Unisys does not support the generation of object modules (use of assembler directive \$OBJ) but will continue to provide full support for the generation of other provided element types which are not object modules (absence of both the \$EXTEND and \$OBJ assembler directives).*

### 10.1. \$OBJ — Select Object Module Output

The \$OBJ directive takes no parameters and causes MASM to produce an object module. The \$OBJ directive must be interpreted before the beginning of the second pass of the main assembly.

With a \$OBJ directive in effect, the resultant output field of the processor call line (`@MASM, options source-input, resultant-output`) is used to name the object module element.

### 10.2. Bank Groups

When creating an object module, MASM also defines a single bank group. This bank group is given the same name as the object module. All text generated in the assembly (if any) is placed in this bank group.



## 10.3. Banks (Location Counters)

All text is organized into banks. In MASM, banks are synonymous with location counters. Each location counter used in the assembly causes a bank to be created. All words generated under a particular location counter are placed in that bank. As with relocatable output, MASM uses location counters to control output. A bank can be void, that is, having no text words.

### 10.3.1. Bank Attributes

Associated with each bank are attributes which determine its use. As the attributes in the following subsections are described, refer to 10.3.2 to see how they are applied. The default attributes for banks are described in 10.3.4.

The zero-fill attribute can be used to indicate whether the bank should be initialized to zero and can be selected by parameter  $e_0$  on the \$BANK directive.

The bank type, owner access, other access, mode, locality, access control, storage, alignment, Interactive Scientific Processor (ISP) local priority, and merge order attributes are all specified as part of parameter  $e_1$  on the \$BANK directive. Subsection Figure 6-1 lists the standard attribute codes supplied with OM\$DEF and describes how a user-defined composite value can be built. These attributes are given as coded numbers whose symbolic names are defined in OM\$DEF.

The address attributes can be specified as parameters  $e_2$ ,  $e_3$ ,  $e_4$  on the \$BANK directive.

The attributes for paging status, hash checking level, common block-aligned, segmented bank, and already zero-filled can all be specified as part of parameter  $e_5$  on the \$BANK directive. The \$FORM OM\$ATTR\_FORM (see 10.8) is provided in OM\$DEF to aid in building composite values of these attributes. These attributes are given as coded numbers whose symbolic names are supplied in OM\$DEF.

The OM\$UNDEFINED value for an attribute is used to postpone definition of the value. (There is one exception: for the bank merge order attribute, the value corresponds to the definition of ANY.) In most cases, the attribute must be defined before load time. This can be done by use of the Linking System command language.

### 10.3.1.1. Bank Type Attribute

The bank type attribute defines what type of usage the bank sees. The attribute names are as follows:

OM\$CODE

The bank contains executable code (instructions) and possibly data.

OM\$COMMON

The bank is a common block.

OM\$DATA

The bank contains data other than a common block.

OM\$GENERAL

The bank contains some combination of executable code and data.

OM\$LINK\_VECTOR

The bank contains link vectors, tables that are necessary for dynamic linking.

OM\$SDD

The bank is a symbolic debugging dictionary bank, containing information necessary for program debugging by the programmers advanced debugging system (PADS).

OM\$UNDEFINED

Bank type temporarily has no attribute.

### 10.3.1.2. Access Permission Attribute

Access to the bank is divided into owner access and other access.

Owner access (special access permission) determines the types of access to the bank that an activity can have. It determines accessing privileges when the accessing activity's key matches the accessed bank's lock.

Other access (general access permission) determines the types of access to the bank that an activity can have. It determines accessing privileges when the accessing activity's key does not match the accessed bank's lock.

OM\$EXECUTE\_ONLY

You can execute the bank but cannot read from or write into it

OM\$GENERAL

You can read from, write into, or execute the bank.

**OM\$NONE**

You cannot read, write, or execute the bank. Access is available only by use of owner access or by use of a gate. (Attribute applies to other access only.)

**OM\$READ\_EXECUTE**

You can read from the bank or execute it but cannot write into it.

**OM\$READ\_ONLY**

You can read from the bank but cannot write into or execute from it.

**OM\$READ\_WRITE**

You can read from or write into the bank but cannot execute it.

**OM\$UNDEFINED**

Access temporarily has no attribute value.

**10.3.1.3. Mode Attribute**

The mode attribute determines the execution mode associated with the bank. Mode determines the storage range available for addressing, the addressing model used, the number of physical banks that can be visible simultaneously, and the available instruction set.

**OM\$BASIC**

The bank is a basic mode code bank or is associated with a basic mode code bank.

**OM\$EXTENDED**

The bank is an extended mode code bank or is associated with an extended mode code bank, so that extended mode addressing is correct for it.

**OM\$ISP**

The bank is an Integrated Scientific Processor (ISP) code bank or is associated with an ISP code bank, so that ISP addressing is correct for it.

**OM\$OTHER**

Reserved for future considerations.

**OM\$UNDEFINED**

Mode temporarily has no attribute value.

### 10.3.1.4. Locality Attribute

The locality (sharing level) attribute defines the level at which the bank is shared; that is, whether it is private to an activity, shared among the activities of a program, shared among the runs accessing a particular application, or shared throughout the computer system. Locality is synonymous with sharing level. It determines a bank's lifetime, that is, whether an existing copy of the bank is used or a new copy is created when an activity first references the bank, and how long the copy exists. The sharing levels are listed from most global to most local.

**Note:** *An activity is defined as a virtual central processing unit (CPU); in other words, an executing program that the Exec treats by multiprogramming as if it were the only program running on the CPU. An activity is the sequence or thread of execution that is the basis of all program execution. MASM and some high-level languages allow a single program to fork into different activities.*

#### OM\$CALLER\_LOCAL

Reserved for future considerations.

#### OM\$SYSTEM

The SYSTEM sharing level means all activities in the system can simultaneously share the physical bank. That is, different runs in different application groups can share the physical bank. Only one copy of the bank exists. It exists until explicitly deleted or until a system failure forces recreation of all banks.

#### OM\$APPLICATION

The APPLICATION sharing level means all activities in an application group can simultaneously share the physical bank. That is, different runs in the application group can share the physical bank, but runs in different application groups cannot. If different application groups use an object module element with application level banks, separate copies of the banks exist in each application group. A copy of an APPLICATION level bank exists until explicitly deleted or until a system failure forces re-creation of all banks. Common banks are shared at the APPLICATION level.

### OM\$RUN

Reserved for future considerations.

### OM\$PROGRAM

The PROGRAM sharing level means all activities in a single executing program can simultaneously share the physical bank. PROGRAM level banks cannot be accessed outside the program that created them. If different runs use an object module element having PROGRAM level banks, separate copies of the banks exist in each run. A copy of a PROGRAM level bank exists until the program that created it terminates.

### OM\$ACTIVITY

The ACTIVITY sharing level means the physical bank is available only to the activity that created it. Each activity using an object module element having ACTIVITY level banks has its own copy of the banks. The bank copy exists until explicitly deleted or until the activity owning the bank terminates.

### OM\$UNDEFINED

Locality temporarily has no attribute value.

## 10.3.1.5. Access Control Attribute

The access control (ring number) attribute, assuming different subsystems, determines whether an activity (executing program; see 10.3.1.4) has special access permission (owner access) or general access permission (other access) to a bank. Access control is synonymous with a hardware ring number. If an activity's access privilege (ring number) value is less than the accessed bank's access control (ring number) value, the activity gets special access permission to the bank. If not, the activity has general access permission. Values for access control are relative to each other, on a continuum from most restrictive (lowest ring number) to least restrictive (highest ring number).

### OM\$KERNEL

The value that represents the most privileged ring number and thus most restrictive lock on a bank. It implies that any activity is restricted to general access permission (other access) to the bank, unless the activity is executing within the same subsystem (that is, has the same domain number).

This value implies that the bank contains the most privileged part of the Exec, providing low level access to system resources and security validation.

Equivalent to a ring number of 0.

### OM\$SHELL

The value that represents the second most-restrictive lock on a bank. It implies that the bank contains the remaining portions of the Exec or very highly privileged portions of the operating system, providing higher level interfaces (for example, to symbionts).

Equivalent to a ring number of 1.

### OM\$TRUSTED

The value that represents the third most-restrictive lock for a bank. It implies that the bank contains trusted and shared portions of the operating system or applications-level software.

Equivalent to a ring number of 2.

### OM\$ORDINARY

The value that represents the fourth most-restrictive lock for a bank. It implies that the bank contains ordinary, nonprivileged software.

Equivalent to a ring number of 3.

### OM\$PUBLIC

The value that represents the least privileged ring number and thus the least restrictive lock for a bank. It applies to the least privileged software and implies that an executing program can always have special access permission (owner access) to the bank.

Equivalent to a ring number of 4.

### OM\$UNDEFINED

Access control temporarily has no value.

### **10.3.1.6. Storage Attribute**

The storage attribute allows control over the type of main storage in which the bank can be loaded. This lets you place specific banks into specific local or main storage devices for special attached processors or explicitly controlled multilevel storage systems. The possible values are as follows:

OM\$HPSU

High-performance storage unit (HPSU)

OM\$ISP\_LOCAL

ISP local storage unit

OM\$MSU

Main storage unit (MSU)

OM\$UNDEFINED

Unspecified

OM\$XSU

Either HPSU or MSU

### **10.3.1.7. Alignment Attribute**

The bank can be placed at a given word boundary by the alignment attribute. The alignment attribute is a 6-bit integer used as the power of two, giving the word alignment boundary. For example, use alignment = 0 ( $2^0 = 1$ ) for single-word alignment and alignment = 1 ( $2^1 = 2$ ) for double-word alignment.

### **10.3.1.8. ISP Local Priority Attribute**

If a bank mode of OM\$ISP for the Integrated Scientific Processor (ISP) and a main storage type of OM\$ISP\_LOCAL (ISP local storage unit) are specified, the bank's priority for local storage can be supplied. The ISP local priority attribute is a 6-bit integer.

### 10.3.1.9. Merge Order Attribute

The merge order attribute determines whether a bank can be merged with any other banks and, if so, the order in which it can be merged. The possible values are

OM\$FIRST

Merge this bank first.

OM\$LAST

Merge this bank last.

OM\$NONE

Do not merge this bank.

OM\$UNDEFINED

Merge this bank in any order. This is the default. This corresponds to the Linking System (static linking) name ANY.

### 10.3.1.10. Address Attribute

The initial lowest relative address for a bank can be explicitly selected by the lower address attribute (parameter  $e_2$  on the \$BANK directive).

The smallest lower address a bank can dynamically expand down to during execution can be specified as parameter  $e_3$ . This minimum address must not exceed the lower address.

The maximum address a bank can dynamically expand up to during execution can be specified as parameter  $e_4$ . The maximum address must be greater than or equal to the lower address + size.



### **10.3.1.11. Paging Status Attribute**

The paging status attribute defines the type of paging desired and can have the following values:

#### **OM\$PAGED**

Any reference to the bank generates a page address translation, or page fault.

#### **OM\$PAGE\_LOCKED**

After page address translation occurs, the entire bank is loaded into memory and locked so that the bank remains loaded until the run is finished.

#### **OM\$UNDEFINED**

The bank does not have any paging address translation performed when referenced. This is the default.

### **10.3.1.12. Hash Checking Level Attribute**

The hash checking level attribute defines the type of checking level that should be done for each instance of a common block.

#### **OM\$HASH1**

Compare the initial value hash code.

#### **OM\$HASH2**

Compare sizes of all instances of the common block.

#### **OM\$HASH3**

Compare both the sizes and initial value hash codes.

#### **OM\$HASH4**

Ensure that the initializing instance is the largest.

#### **OM\$HASH5**

Ensure that the initializing instance is the largest, and that the hash codes match.

#### **OM\$UNDEFINED**

No checking. This is the default.

### 10.3.1.13. Additional Attributes

The following attributes can also be specified as part of  $e_5$ :

OM\$CB\_ALIGNED

Specifies that the common block requires word boundary alignment.

OM\$FLAT

Specifies that the bank is to be loaded into the FLAT space.

OM\$SEGMENTED

Specifies that the bank is created by acquiring as many 262K segments as needed to complete the bank. The 262K segments must have contiguous bank descriptor indexes (BDI), but are not required to be physically adjacent. This differs from large banks, which must be contiguous in physical memory.

OM\$ZEROED

Specifies that the bank has already been fully initialized.

### 10.3.2. \$BANK Directive — Define Attributes for a Bank

The format of the \$BANK directive is as follows:

```
$BANK[ ,  $e_0$ ]  $e_1$ [ ,  $e_2$ [ ,  $e_3$ [ ,  $e_4$ [ ,  $e_5$ ]]]] [  $e_6$ [ ,  $e_7$ ]]
```

where:

$e_0$

is the zero-fill attribute, and the default value is 0 (off).

$e_1$

is the coded value describing attributes.

$e_2$

is the initial lowest address, and the default address is 0.

$e_3$

is the minimum address, and the minimum address default is 0.

$e_4$

is the maximum address, and the maximum address default is 262K.

$e_5$

is the coded value describing additional attributes.

$e_6$

is the bank name. The default name is formed from the user output element name and the location counter as follows: *output-element-name\$lc*.

$e_7$

is the related symbolic debugging dictionary (SDD) bank, and the default is none.

Each subfield is optional except  $e_1$ . Attributes apply to the bank defined by the current location counter. For a complete description of each attribute, see 10.3.1. A short description of each parameter follows.

Parameter  $e_0$  is a conditional setting for the zero-fill attribute. If  $e_0$  is unspecified or 0, zero-fill is not performed. To request a bank to be zero-filled, set  $e_0$  to 1. Other values for  $e_0$  should be considered undefined.

Parameter  $e_1$  is an integer value without relocation that specifies several bank attributes. The attributes specified in  $e_1$  are as follows:

## Object Modules

---

Attribute	Field Size
Bank type	6 bits
Owner access	6 bits
Other access	6 bits
Mode	6 bits
Locality	6 bits
Access control	6 bits
Storage class	6 bits
Alignment	6 bits
ISP local priority	6 bits
Merge order	6 bits
Unused	12 bits

These attributes are coded as numbers in successive fields of a 72-bit number, which is subdivided by the following form.

OM\$BANK\_FORM      \$FORM      6,6,6,6,6,6,6,6,6,6,6,12

**Note:** *The last field is unused and should be zero for future compatibility. This form and symbolic names for the attributes are defined in OM\$DEF. See 10.3.1 for a description of the attributes. See 10.8 for a list of the standard composite values of the attributes contained in OM\$DEF.*

Parameters  $e_2$  through  $e_4$  define the address space of the bank and, if given, must be integer numbers without relocation.

Parameter  $e_5$  is an integer value without relocation that specifies several additional bank attributes. The attributes specified in  $e_5$  are as shown in the following table:

Attribute	Field Size
Unused	1 bit
Unused	1 bit
Flat address space	1 bit
Common block-aligned	1 bit

continued

Attribute	Field Size
Already zero-filled	1 bit
Segmented bank	1 bit
Paging status	6 bits
Hash checking level	6 bits
Unused	18 bits

These attributes are coded as numbers in successive fields of a 36-bit number, which is subdivided by the following form:

```
OM$ATTR_FORM      $FORM      1,1,1,1,1,1,6,6,18
```

**Note:** *The first two fields and the last field are unused and should be zero for future compatibility. This form and symbolic names for the attributes are defined in OM\$DEF. See 10.3.1 for a description of the attributes.*

The bank name,  $e_6$ , can be any string chosen from the ASCII character set. Refer to 10.3.3 for a description of the bank name.

The related symbolic debugging dictionary (SDD) bank,  $e_7$ , should be an integer value without relocation, which specifies the location counter of the SDD bank.

### 10.3.3. Bank Name

The default bank name for each location counter is OUTPUT\_ELEMENT\_NAME\$, followed by the location counter number expressed in decimal using two digits (for example, OM\$01). Banks can be explicitly named on the \$BANK directive (see 10.3.2). The names have no significance to MASM and can be any ASCII string. As with relocatable output, MASM uses location counters to control output generation.

### 10.3.4. Default Values for Banks

Banks should not be allowed to take default attributes. This is because the assembler has no way to distinguish a code bank from a data bank, and general purpose banks are not initially loaded. MASM assumes that odd location counters contain code and that even location counters contain data and defaults.

MASM also tries to determine if the bank is a basic or an extended mode bank by defaulting to the mode under which the location counter is first referenced. Location counter 0 is the default location counter at the start of all assemblies. It is for this reason that location counter 0 always defaults to basic mode, since the location counter is initialized before any mode is established by a \$BASIC or \$EXTEND directive. MASM deletes location counter 0 at the end of the assembly unless one of the following is true:

- A \$(0) occurs within the main assembly.
- Location counter 0 is not void.
- Location counter 0 is explicitly defined via the \$BANK directive.

Banks with the default attributes can still be inadequate for most users in many cases. The procedure OM\$USE\_LV is used to explicitly declare the link vector bank.

### 10.3.5. \$BANK Example

This example shows the definition of standard code (instruction), data, and link vector banks. Two instruction banks are created: IBANK, defined by location counter 1, and SUB\_I, defined by location counter 3. Both instruction banks start at 01000. A data bank is created, named DBANK. This bank is defined by location counter 0, and starts at 040000. A link vector bank, named LVBANK, is defined by location counter 4 and starts at 0. The standard bank attributes, OM\$CODE\_EMB, OM\$DATA\_EMB, and OM\$LV\_EMB, are defined in OM\$DEF. See 10.8 for the definition of the attribute values.

```

$ASCII
$INCLUDE 'MAXR$'
$INCLUDE 'OM$DEF'
$INCLUDE 'EM2CB$P'
$EXTEND
$OBJ
$(4) $BANK OM$LV_EMB,000000 'LVBANK'
...
$(0) $BANK OM$DATA_EMB,040000 'DBANK'
...
$(3) $BANK OM$CODE_EMB,01000 'SUB_I'
...
$(1) $BANK OM$CODE_EMB,01000 'IBANK'
...
$END
```

## 10.4. References

External entities can be imported by the \$IMPORT directive to describe how the name is resolved. If no \$IMPORT directive appears for an external reference, default attributes are applied (see 10.4.4). When the name of the imported item is used in an expression, its attributes are used to build a reference expression for the text word.

### 10.4.1. Reference Attributes

In addition to the name of a reference, which can be any ASCII string, the MASM programmer can also specify the reference type, strength of the reference, the resolution type, storage type control, and conformance to the standard calling sequence (SCS) conventions. These attributes can all be specified as part of parameter  $e_1$  on the \$IMPORT directive.

Subsection 10.8 lists the standard composite values of these attributes, which are defined in OM\$DEF. These attributes are given as coded numbers whose symbolic names are defined in OM\$DEF.

#### 10.4.1.1. Reference Type Attribute

Reference type defines what kind of definition this reference should resolve to. The reference types are as follows:

OM\$ANY

Resolves to any type of external definition, other than common block.

OM\$CODE

Resolves to an external routine entry point.

OM\$COMMON

Resolves to a common block definition.

OM\$CONSTANT

Resolves to a constant value.

OM\$DATA

Resolves to a data item external definition.

OM\$UNDEFINED

Reference type temporarily has no attribute value.

### 10.4.1.2. Strength Attribute

The strength attribute controls when the reference is resolved. Available strength attributes are the following:

OM\$NORMAL

Resolved when it is first encountered during execution.

OM\$STATIC

Resolved at static link. This attribute applies only to logical bank number (LBN) resolution types (see 10.4.1.3).

OM\$STRONG

Resolved when the bank that contains it is loaded.

OM\$UNDEFINED

This attribute type temporarily has no value.

### 10.4.1.3. Resolution Type Attribute

Resolution type defines the information desired of the reference. Most references have a resolution type of OM\$BDI\_OFFSET or OM\$BDI\_VA. Resolution type OM\$LVE is used only in a link vector bank. The other values are provided for special purposes. The following resolution types are available:

OM\$BDI

Address tree level and bank descriptor index (BDI)

OM\$BDI\_OFFSET

Offset from bank

OM\$BDI\_VA

Both BDI and offset from bank

OM\$ELT\_INFO

Element information indicates whether the element that contains the resolved reference is a zero overhead object module (ZOOM) or object module.

OM\$ENTRY\_VAR

The reference is to a UCS FORTRAN entry variable, resolution is two link vector entries.

OM\$EP\_VA

An externally referenced code entry point 36-bit virtual address without a latent parameter or creation of a gate.



OM\$LBN

Logical bank number (LBN).

OM\$LBN\_OFFSET

Offset from LBN.

OM\$LBN\_VA

Both LBN and offset.

OM\$LVE

Link vector entry.

OM\$OM\_ID

The unique identifier of the object module identifier.

OM\$UNDEFINED

This type temporarily does not have an attribute.

### 10.4.1.4. Storage Attribute

The storage attribute specifies the storage attribute associated with the reference. This means that the reference must resolve to a definition in a bank with the same storage attribute as the one attached to the reference. The possible values for the storage attribute are as follows:

OM\$HPSU

High-performance storage unit (HPSU)

OM\$ISP\_LOCAL

ISP local storage unit

OM\$MSU

Main storage unit (MSU)

OM\$UNDEFINED

Storage temporarily has no attribute value.

OM\$XSU

Either HPSU or MSU

### 10.4.1.5. SCS Conformance Attribute

An attribute is provided to specify conformance with the standard calling sequence (SCS) conventions. The available values for this attribute are as follows:

- OM\$LOOSE
- OM\$NONE (default)
- OM\$STRICT

## 10.4.2. \$IMPORT Directive — Define Attributes for a Reference

The format of the \$IMPORT directive is as follows:

```
label $IMPORT e1[,e2] [e3]
```

where:

$e_1$

is the coded attribute bits in a 72-bit number.

$e_2$

is the library code name and the default value is none.

$e_3$

is the XREF name or value to which attributes apply.

The \$IMPORT directive applies special Linking System attributes to the value or name given and creates a new value. It also allows referencing an external entity whose name does not conform to MASM syntax. Parameters  $e_2$  and  $e_3$  are optional.

Parameter  $e_1$  is an integer value without relocation that specifies the reference attributes. The value is 72 bits long and subdivided into six fields. The first five fields are 6 bits each and specify the reference type, strength, resolution type, storage class, and SCS conformance. The last field is 42 bits long and is unused and should be set to 0. A MASM \$FORM named OM\$REF\_FORM (see Figure 6-1) is provided in OM\$DEF to help assemble this value. See 10.4.1 for more attribute information.

Parameter  $e_2$  is a string that names the library search chain. If parameter  $e_3$  is relocated by a location counter,  $e_2$  must be either omitted or a null string. For external references, parameter  $e_2$  is optional. See 10.4.3 for more information on library search chains.

Parameter  $e_3$ , if given, must be either an integer value with simple relocation or a string. If an integer value is given, the resultant value has the absolute part and relocation of the value (either by LC or XREF). If a string is given, the resultant value has relocation by external reference whose name is the string value and the absolute part is zero. If parameter  $e_3$  is not specified, the resultant value is relocated by an external reference whose name is the same as the label given, and the absolute part is zero. In any case, the values in  $e_1$  and  $e_2$  (the library code name), if specified and valid, are used. If  $e_3$  specifies values with multiple relocation subitems, an R flag is issued. If  $e_3$  specifies values with relocation attributes or library search chain, they are replaced either by the values on the directive or by nothing.

### 10.4.3. Library Search Chains

The linking system has the ability to search many files to resolve an undefined name. Which files to search can be specified in a list of files known as the library search chain. Library search chains are named by the library code name on the \$IMPORT directive, to bind them to a reference. Only the code name is given in the assembly code; the chain itself is defined outside of MASM. The name can be any string from the ASCII character set. For further information on library search chains, refer to the Linking System documents, especially the Linking System Programming Reference Manual.

### 10.4.4. Default Values for References

The following table shows the MASM reference defaults:

Attribute	Default Value
Name	Label used in the expression
Reference type	'OM\$ANY'
Resolution type	'OM\$BDI_OFFSET'
Strength	'OM\$STRONG'
Storage type	'OM\$MSU'
SCS conformance	'OM\$NONE'

### 10.4.5. \$IMPORT Directive Example

This example imports a data bank address and a code bank address and changes their names. The standard reference attributes, OM\$DATA\_VA and OM\$CODE\_VA, are defined in the library element OM\$DEF to be strong references. Both references resolve to a virtual address reference. No library code name is used. \$IMPORT changes the names of SYSIN and DSTART to READ and IOBUFREF and sets the reference types to OM\$CODE and OM\$DATA.

Subsection 10.8 lists the standard composite values of these attributes, which are defined in OM\$DEF. Compare this example to the \$EXPORT example in 10.5.4.

```

$ASCII
$INCLUDE 'MAXR$'          . Define registers
$INCLUDE 'OM$DEF'         . Define OM values
$INCLUDE 'EM2CB$P'        . Define extended mode ERs
$EXTEND                  . Turn on extended mode
$OBJ                     . Create an object module
READ $IMPORT OM$CODE_VA 'SYSIN' . Define subroutine name
IOBUFREF $IMPORT OM$DATA_VA 'DSTART' . Define IO buffer
.
$(4) $BANK OM$LV_EMB,000000 'LVBANK' . Define link vector bank
IOBUFVA + IOBUFREF . Virtual address of IOBUFREF
READVA + READ . Virtual address of READ
.
$(0) $BANK OM$DATA_EMB,040000 'DBANK' . Define D-bank
...
$(3) $BANK OM$CODE_EMB,01000 'SUB_I' . Define I-bank
...
$(1) $BANK OM$CODE_EMB,01000 'IBANK' . Define I-bank
...
$END .

```

# 10.5. Definitions

Definitions and entry points can be exported by the \$EXPORT directive. If no \$EXPORT directive appears for an external definition, default attributes are applied (see 10.5.3).

## 10.5.1. Definition Attributes

In addition to the name, which can be any ASCII string, definitions also have a definition type, a visibility attribute, an SCS conformance attribute, and an address type attribute. The following attributes can be specified as part of parameter  $e_2$  on the \$EXPORT directive. Subsection 10.8 lists the standard composite values of these attributes, which are defined in OM\$DEF. These attributes are given as coded numbers whose symbolic names are defined in OM\$DEF.

### 10.5.1.1. Definition Type Attribute

The definition type is used by the Linking System to qualify the name (that is, distinguish between names) and must match the type of the reference to the name. The possible definition types are as follows:

OM\$CODE

Entry point address of an external routine

OM\$COMMON

Common block address

OM\$CONSTANT

Constant value

OM\$DATA

Data item address

OM\$FIXED\_GATE

Entry point address for a fixed gate

OM\$UNDEFINED

Definition type temporarily has no attribute value.

### 10.5.1.2. Visibility Attribute

The scope of the definition is defined by the visibility attribute, which can be either of the following:

OM\$EXTERNAL

Known outside the object module

OM\$INTERNAL

Not known to other object modules

OM\$SS\_EXTERNAL

An external entry point definition for a fixed gate

### 10.5.1.3. SCS Conformance Attribute

An attribute is provided to specify conformance with the standard calling sequence (SCS) conventions. The available attributes are as follows:

- OM\$LOOSE
- OM\$NONE (default)
- OM\$STRICT

### 10.5.1.4. Address Type Attribute

The address type attribute is provided to specify the format of the address constant. The available values are as follows:

#### OM\$EIGHT\_LVL\_VA

The address constant is an extended mode virtual address in (L,BDI,OFFSET) format, where L = 3 bits, BDI = 15 bits and OFFSET = 18 bits.

#### OM\$FOUR\_LVL\_VA

The address constant is an extended mode virtual address in (L,BDI,OFFSET) format, where L = 2 bits, BDI = 16 bits and OFFSET = 18 bits.

#### OM\$TWO\_LVL\_VA

The address constant is a basic mode virtual address in (E,BDI,OFFSET) format.

#### OM\$UNDEFINED

A 36-bit fixed constant (default). If the value is a virtual address, its format is unknown.



## 10.5.2. \$EXPORT Directive — Define Attributes for a Definition

The format of the \$EXPORT directive is as follows:

```
$EXPORT       $e_1, e_2, e_3$ 
```

where:

$e_1$

is the external definition name.

$e_2$

is the coded attribute words.

$e_3$

is the value of definition.

The first parameter,  $e_1$ , defines the name of the definition, that is, the name by which this entity is known outside of this assembly. The name subfield has two distinct forms. It can be a character string expression (other than a simple variable), or it can be a simple variable that represents a value (that is, integer with or without relocation, string, or floating-point; not control information like \$PROC or \$FUNC names or forms; also, forms on values are not retained).

If the name is a string expression, it is used as the name of the definition and need not conform to MASM syntax for names. In this case, the value subfield,  $e_3$ , is required and can be any arbitrary value expression as just described.

If the name is a simple variable, then its name is also the definition name and its value is the definition value. In this case, the value subfield,  $e_3$ , is redundant and illegal.

The second parameter,  $e_2$ , defines the definition type, its visibility, and its SCS conformance, using the top four 6-bit fields of a 72-bit value. The remainder of the value is reserved for future expansion and should be set to zero. A MASM \$FORM named OM\$DEF\_FORM (see 10.8) and the attribute names equated to their values are in OM\$DEF to make it easier to assemble this value. See 10.5.1 for more attribute information.

### 10.5.3. Default Values for Definitions

The following table shows the MASM definition default values:

Attribute	Default Value
Definition type	'OM\$CODE', 'OM\$CONSTANT', 'OM\$DATA', or 'OM\$UNDEFINED' (depending on the value)
Visibility	'OM\$EXTERNAL'
SCS conformance	'OM\$NONE'
Address type	'OM\$UNDEFINED'

### 10.5.4. \$EXPORT Directive Example

This example defines a code entry point and an external data definition to be used in the same context as the \$IMPORT example (see 10.4.5). See 10.5.1 for definition of attribute values.

```

$ASCII
$INCLUDE 'MAXRS' . Define registers
$INCLUDE 'OM$DEF' . Define OM values
$INCLUDE 'EM2CB$P' . Define extended mode ERs
$EXTEND . Turn on extended mode
$OBJ . Create an object module
READ $IMPORT OM$CODE_VA 'SYSIN' . Define subroutine name
IOBUFREF $IMPORT OM$DATA_VA 'DSTART' . Define IO buffer
.
$(4) $BANK OM$LV_EMB,000000 'LVBANK' . Define link vector bank
IOBUFVA + IOBUFREF . Virtual address of IOBUFREF
READVA + READ . Virtual address of READ
.
$(0) $BANK OM$DATA_EMB,040000 'DBANK' . Define D-bank
DSTART .
$EXPORT DSTART,OM$DATA_DEF . Make DSTART an external reference
...
$(3) $BANK OM$CODE_EMB,01000 'SUB_I' . Define I-bank
$EXPORT SYSIN,OM$ENTRY_DEF . Make SYSIN an external entry point
SYSIN .
...
$(1) $BANK OM$CODE_EMB,01000 'IBANK' . Define I-bank
...
$END

```

## 10.6. Link Vectors

The usual method of resolving external references involves the use of a data structure known as the link vector. The link vector is a location counter whose text is consecutive link vector entries. This location counter should not contain any other text. A link vector entry is a reference to a name and has a resolution type (see 10.4) equal to OM\$LVE. The location counter must be declared to be a link vector bank; that is, the bank type (see 10.3) is set to OM\$LINK\_VECTOR. The \$BANK directive (see 10.3.2) and the \$IMPORT directive (see 10.4.2) are used to assign these attributes to the bank and to the reference, respectively.

The initial value of a link vector entry is not the address of the reference, but a value that causes a special contingency interrupt in the Exec. The Exec recognizes this interrupt (called a link fault) and invokes the Linking System with the relocation information to resolve the name and get the address (L,BDI,Offset) of the target. This value is placed in the link vector, and the instruction(s) that provoked the link fault are re-executed. In basic mode, the system backs up two instructions before the LBJ instruction. In extended mode, only CALL or LBU is repeated.

### 10.6.1. Basic Mode

When a routine is entered in basic mode, either as the main program or as a subroutine called using a common coding sequence, register X3 is pointing to the base of the link vector bank. For calling external subroutines, it is necessary to move this value to another index register (typically X2). To use the link vector entry, a common coding sequence must be used. For code references (transfers of control), the sequence is as follows:

```
LX    X11,offset,x_reg
LX    X3,offset+1,x_reg
LX    X11,0,X11
```

*offset* is the offset into the link vector of the link vector entry, and *x\_reg* is an index register (other than X0, X3, or X11) that points to the base of the link vector.

For data references in basic mode, the code sequence is as follows:

```
LX      X11,offset,x_reg
TNZ     offset+1,x_reg
LBJ     X11,0,X11
LBJ     X11,$+1
LX      x_reg_2,offset,x_reg
```

Again, *offset* is the offset of the link vector entry and *x\_reg* points to the start of the link vector. The index register *x\_reg\_2* is loaded with the address of the external reference and can be used as the base of that bank.

### 10.6.2. Extended Mode

In extended mode, link vector entries are two words long. A base register is needed to base the link vector bank, and this register should normally be preserved across a call. When a routine is entered in extended mode, either as the main program or as a subroutine called using the standard calling sequence, register R0 is pointing to the base of the link vector bank. Code references (transfers of control) are as follows:

```
CALL lve
```

lve is the link vector entry of the target routine. The lve is addressed using the usual extended mode addressing with base, displacement, and optional index register. CALL is an indirect jump, and the link vector is the address of the target routine. Data references require loading a base register from a link vector entry, as in the following example:

```
LBU b,lve
```

In this example, b is the base register to load, and lve is the address of the link vector entry using base, displacement, and optional index register. The loaded base register can then be used to access the data.

### 10.6.3. Extended Mode Example

The following example shows the extended mode of the CALL and LBU instructions to be used with the previous examples:

```

$ASCII
$INCLUDE      MAXR$'      . Define registers
$INCLUDE      'OM$DEF'    . Define OM values
$INCLUDE      'EM2CB$P'   . Define extended mode ERs
$EXTEND       . Turn on extended mode
$OBJ          . Create an object module
READ          $IMPORT     OM$CODE_VA  'SYSIN'  . Define subroutine name
IOBUFREF      $IMPORT     OM$DATA_VA  'DSTART' . Define IO buffer
.
$(4)          $BANK        OM$LV_EMB,000000 'LVBANK' . Define link vector bank
IOBUFVA      +            IOBUFREF      . Virtual address of IOBUFREF
READVA       +            READ          . Virtual address of READ
.
$(0)          $BANK        OM$DATA_EMB,040000 'DBANK' . Define D-bank
DSTART       .
$EXPORT       DSTART,OM$DATA_DEF      . Make DSTART an external reference
ENDMSG       '*** END OF FI          LE ***' . Define end message
ENDMSG      $EQU          $-ENDMSG      . Message length
ENDPKT      EM$APRINTPKT  ENDMSG,ENDMSG .
.
IOBUF        $RES          20          . Define IO buffer
EOFFLG       $EQU,F,S2     2,X6,,B3    . Define end of file flag
WRDSRD       $EQU,F,H2     2,X6,,B3    . Define number of words read
WRDSPRT      $EQU,F,S3     1,X7,,B3    . Define number of words to print
ARDPKT       EM$AREADPKT   IOBUF       . Create packet for ER AREAD$
APTPKT       EM$APRINTPKT  BUF,20      . Packet for ER APRINT$
.
$(3)          $BANK        OM$CODE_EMB,01000 'SUB_I' . Define I-bank
$EXPORT      SYSIN,OM$ENTRY_DEF      .
.
SYSIN        .
EM$AREAD     ARDPKT,B3          . Call ER AREAD$ proc
RTN          . Return
.
$(1)          $BANK        OM$CODE_EMB,01000 'IBANK' . Define I-bank
START$*      .
LBU          B2,R0            . B2 = VA of link vector bank
LBU          B3,IOBUFVA-$LCB(4),,B2 . B3 = start of data bank
.
LOOP         .
L,U          X5,READVA-$LCB(4)      . X5 = link vector entry of subroutine
CALL         0,X5,B2              . Call the subroutine
L,U          X6,ARDPKT-$LCB(0)      . X6 = offset of read packet
L,U          X7,APTPKT-$LCB(0)      . X7 = offset of print packet
TZ           EOFFLG              . End of file?
J            FINISH              . Yes, jump to finish
EM$APRINT    APTPKT,B3            . Call ER APRINT$
J            LOOP                 .
.
FINISH       .
EM$APRINT    ENDPKT,B3            . Print end of file message
RTN          .
$END         .

```

### 10.6.4. Standard Procedures for the Creation of Link Vectors

Rather than force a cumbersome coding sequence on the programmer, three MASM procedures exist in the MASM definition element OM\$DEF for initialization, code references, and data references.

OM\$DEF should be placed in SYS\$LIB\$\*MASM or otherwise made available to the assembly (see NO TAG). OM\$DEF is brought into an assembly by \$INCLUDE 'OM\$DEF'; its procedures can be used for either basic or extended mode code.

#### 10.6.4.1. Initialization — OM\$USE\_LV

In basic mode, the initialization procedure is called as follows:

```
OM$USE_LV,lc bankname,x_reg
```

In extended mode, the initialization procedure is called as follows:

```
OM$USE_LV,lc bankname,x_reg,b_reg
```

This is required to establish the link vector bank. The location counter *lc* can be any number from 0 to 63. If the location counter is not given, LC 17 is used. The bank name is a string in the system character set (uppercase and lowercase characters are retained if \$ASCII is on). If no bank name is given, the default bank name is used (see 10.3.4). The index register, if given, is the default link vector base for calls to the reference \$PROC directives that do not have a register given. *b\_reg*, if given, is used only if generating in extended mode and is the base register on which the link vector bank is based. Before a user executes an instruction whose operand is a link vector entry, the user must initialize *b\_reg* by executing the following instruction:

```
LBU b_reg,R0 .
```

Extended mode must be selected before this procedure is called. All link vector entries and references are in the same mode.

#### 10.6.4.2. Code References — OM\$CALL

In basic mode, the code reference procedure is called as follows:

```
OM$CALL prog_name,x_reg lib_code_name  
OM$HV_CALL prog_name,x_reg lib_code_name
```

In extended mode, the code reference procedure is called as follows:

```
OM$CALL prog_name,x_reg,b_reg lib_code_name  
OM$HV_CALL prog_name,x_reg,b_reg lib_code_name
```

This creates a link vector entry and generates the code for the common coding sequence based on whether \$BASIC or \$EXTEND mode is in effect. *Prog\_name* can be an undefined MASM label (no leading '\$', maximum of 12 significant characters, converted to uppercase) or it can be a string in the system character set. If the X register is not given, the one from the OM\$USE\_LV call is used. The X register is assumed to be pointing to the base of the link vector bank. In \$BASIC mode, OM\$CALL generates code using registers X3 and X11, and OM\$HV\_CALL generates code using registers R0 and X11. In \$EXTEND mode, both OM\$CALL and OM\$HV\_CALL generate code using registers R0 and X11. Any other registers from X1 to A3 can be used. In extended mode, a base register must be used to address the link vector. The base register can be given on the reference call or on the initialization call (OM\$USE\_LV). The library code name is a string expression. If no library is named, none is used for the reference.

### 10.6.4.3. Data References — OM\$LOAD\_BDR

In basic mode, the data reference procedure is called as follows:

```
OM$LOAD_BDR    xreg1,xref,xreg2  lib_code_name
```

In extended mode, the data reference procedure is called as follows:

```
OM$LOAD_BDR    breg1,xref,xreg2,breg2  lib_code_name
```

This creates a link vector entry and generates the code to provoke a link fault based on whether \$BASIC or \$EXTEND mode is in effect. *xref* can be either a label or a string like *prog\_name* in OM\$CALL. *xreg2* is the X register pointing to the base of the link vector. If *xreg2* is not given, the index register from the OM\$USE\_LV call is used. Any X register other than X0 or X11 can be used. In extended mode, a base register must be used to base the link vector bank. This base register can be from the OM\$USE\_LV call or from *breg2* on the call to OM\$LOAD\_BDR. In basic mode, code is generated to load *xreg1* with the first word of the link vector and can then be used to reference the data. In extended mode, *breg1* is loaded with the address of the first word of the link vector. *lib\_code\_name* is the same as for OM\$CALL.

### 10.6.4.4. Extended Mode Example

The following example shows the extended mode of the MASM procedures OM\$LOAD\_BDR and OM\$CALL to be used with the previous examples.

```

$ASCII
$INCLUDE      'MAXR$'
$INCLUDE      'OM$DEF'
$INCLUDE      'EM2CB$P'
$EXTEND
$OBJ
.
.
$(4)          OM$USE_LV      'LVBANK',,B2          . Define link vector bank
.
$(0)          $BANK          OM$DATA_EMB,040000 'DBANK' . Define D-bank
DSTART
ENDMSG
ENDMSGGL
ENDPKT
.
.      '*** END OF FILE ***'
.      $EQU                  $-ENDMSG
.      EM$APRINTPKT          ENDMSG,ENDMSGGL
.      Packet for EM$APRINT
.
IOBUF         $RES          20
EOFFLG        $EQUF,S2      2,X6,,B3
WRDSRD        $EQUF,H2      2,X6,,B3
WRDSPRT       $EQUF,S3      1,X7,,B3
ARDPKT        EM$AREADPKT   IOBUF
APTPKT        EM$APRINTPKT  BUF,20
.      Packet for ER APRINT$
.
$(3)          $BANK          OM$CODE_EMB,01000 'SUB_I' . Define I-bank
.
SYSIN
.
.      EM$AREAD              ARDPKT,B3
.      Call ER AREAD$ proc
.      RTN
.      Return
.
$(1)          $BANK          OM$CODE_EMB,01000 'IBANK' . Define I-bank
START$*
.
LBU           B2,R0
.      B2 = VA of link vector bank
OM$LOAD_BDR   B3,DSTART
.      B3 = start of data bank
.
.      OM$CALL              SYSIN
.      Call the subroutine
L,U           X6, ARDPKT-$LCB(0)
.      X6 = offset of read packet
L,U           X7,APTPKT-$LCB(0)
.      X7 = offset of print packet
TZ            EOFFLG
.      End of file?
J             FINISH
.      Yes, jump to finish
LA            A1,WRDSRD
.      Get number of words read and
SA            A1,WRDSPRT
.      put it in the print packet
EM$APRINT     APTPKT,B3
.      Call ER APRINT$
J             LOOP
.
FINISH
.
.      EM$APRINT             ENDPKT,B3
.      Print end of file message
.      RTN
.
$END
.

```



## **10.7. Defaults**

The attribute fields on the \$BANK, \$IMPORT, and \$EXPORT directives have the value zero in any subfield that is not specified. This equates to selecting “OM\$UNDEFINED” (or its equivalent for the attribute), which can be resolved later if necessary. The default values then, treated as a special case in MASM, only apply to banks, references, and definitions for which there is no corresponding \$BANK, \$IMPORT, or \$EXPORT directive. The default values selected provide compatibility with existing code and help new users develop their programs more quickly. Refer to 10.3.4 (\$BANK), 10.4.4 (\$IMPORT), and 10.5.3 (\$EXPORT) for default attribute values.

### 10.8. Library Definition Element

Attribute codes are defined for banks, references, and definitions in a MASM element named OM\$DEF. The names used in the previous descriptions of the attributes are the names defined equal to the proper value in the element.

The following \$FORMs are useful for the \$BANK, \$IMPORT, and \$EXPORT directives:

OM\$BANK_FORM	\$FORM	6,6,6,6,6,6,6,6,6,12
OM\$ATTR_FORM	\$FORM	1,1,1,1,1,1,6,6,18
OM\$REF_FORM	\$FORM	6,6,6,6,6,42
OM\$DEF_FORM	\$FORM	6,6,6,6,48

Standard bank declarations are predefined in OM\$DEF, as shown in the following example:

OM\$CODE_EMB	\$EQU	+(OM\$BANK_FORM	. Define a code bank
		OM\$CODE,;	. Type
		OM\$READ_EXECUTE,;	. Owner access
		OM\$NONE,;	. Other access
		OM\$EXTENDED,;	. Mode
		OM\$PROGRAM,;	. Locality
		OM\$PUBLIC,;	. Access control
		OM\$MSU,;	. Storage
		0,;	. Alignment
		0,;	. ISP local priority (unused)
		OM\$UNDEFINED)	. Merge order (any)

Similar definitions exist for the bank types shown in the following table:

Bank Declarations	Attributes in OM\$BANK_FORM Order
OM\$DATA_EMB	OM\$DATA, OM\$READ_WRITE, OM\$NONE, OM\$EXTENDED, OM\$ACTIVITY, OM\$PUBLIC, OM\$MSU, 0, 0, OM\$UNDEFINED
OM\$LV_EMB	OM\$LINK_VECTOR, OM\$READ_WRITE, OM\$NONE, OM\$EXTENDED, OM\$ACTIVITY, OM\$PUBLIC, OM\$MSU, 0, 0, OM\$UNDEFINED
OM\$COMMON_EMB	OM\$COMMON, OM\$READ_WRITE, OM\$NONE, OM\$EXTENDED, OM\$PROGRAM, OM\$PUBLIC, OM\$MSU, 0, 0, OM\$UNDEFINED
OM\$CODE_BMB	OM\$CODE, OM\$READ_EXECUTE, OM\$NONE, OM\$BASIC, M\$PROGRAM, OM\$PUBLIC, OM\$MSU, 0, 0, OM\$UNDEFINED
OM\$DATA_BMB	OM\$DATA, OM\$GENERAL, OM\$NONE, OM\$BASIC, OM\$ACTIVITY, OM\$PUBLIC, OM\$MSU, 0, 0, OM\$UNDEFINED
OM\$LV_BMB	OM\$LINK_VECTOR, OM\$GENERAL, OM\$NONE, OM\$BASIC, OM\$ACTIVITY, OM\$PUBLIC, OM\$MSU, 0, 0, OM\$UNDEFINED
OM\$COMMON_BMB	OM\$COMMON, OM\$GENERAL, OM\$NONE, OM\$BASIC, M\$PROGRAM, OM\$PUBLIC, OM\$MSU, 0, 0, OM\$UNDEFINED

The available standard definition attribute definitions are as follows:

OM\$ENTRY_DEF	\$EQU	+(OM\$DEF_FORM ; OM\$CODE,; OM\$EXTERNAL,; OM\$NONE,; OM\$UNDEFINED,;	. Define an entry point . Type . Scope . SCS conformance . Address type
OM\$DATA_DEF	\$EQU	+(OM\$DEF_FORM ; OM\$DATA,OM\$EXTERNAL,OM\$NONE,; OM\$UNDEFINED)	. Data . definition
OM\$CONST_DEF	\$EQU	+(OM\$DEF_FORM ; OM\$CONSTANT,OM\$EXTERNAL,OM\$NONE,; OM\$UNDEFINED)	. External . constant
OM\$FG_DEF	\$EQU	+(OM\$DEF_FORM ; OM\$FIXED_GATE,OM\$EXTERNAL,; OM\$NONE,OM\$FOUR_LVL_VA)	. Fixed gate definition
OM\$CBEP_DEF	\$EQU	+(OM\$DEF_FORM ; OM\$CODE,OM\$EXTERNAL,; OM\$NONE,OM\$FOUR_LVL_VA)	. Common block definition

## Object Modules

---

The available standard reference attribute definitions are as follows:

OM\$CODE_REF	\$EQU	+(OM\$REF_FORM OM\$CODE,OM\$STRONG,; OM\$BDI_OFFSET,OM\$MSU,OM\$NONE)	. Code . reference
OM\$DATA_REF	\$EQU	+(OM\$REF_FORM OM\$DATA,OM\$STRONG,; OM\$BDI_OFFSET,OM\$MSU,OM\$NONE)	. Data . reference
OM\$CODE_LVE	\$EQU	+(OM\$REF_FORM OM\$CODE,OM\$NORMAL,; OM\$LVE,OM\$MSU,OM\$NONE)	. Link vector . for code
OM\$DATA_LVE	\$EQU	+(OM\$REF_FORM OM\$DATA,OM\$NORMAL,; OM\$LVE,OM\$MSU,OM\$NONE)	. Link vector . for data
OM\$CODE_VA	\$EQU	\$REF_FORM OM\$CODE,OM\$STRONG,; OM\$BDI_VA,OM\$MSU,OM\$NONE)	. 36-bit code . virtual address
OM\$DATA_VA	\$EQU	+(OM\$REF_FORM OM\$DATA,OM\$STRONG,; OM\$BDI_VA,OM\$MSU,OM\$NONE)	. 36-bit data . virtual address
OM\$ENTRY_VA	\$EQU	+(OM\$REF_FORM OM\$CODE,OM\$STRONG,; OM\$EP_VA,OM\$MSU,OM\$NONE)	. 36-bit virtual . address without . a gate

You can define other values as needed.

## 10.9. Compatibility

The following discussion describes some areas where existing code must be changed to convert it to the object module environment. Though not a complete list of changes, it is intended to help managers and designers anticipate conversion efforts. See Appendix B for changes that have been required of MASM as the object module environment evolves.

### 10.9.1. \$INFO Directives

The MASM \$INFO directive passes special control information to the Collector. Most of this information has a counterpart in the object module environment, although the mechanisms have been changed. These \$INFO directives have to be rewritten, using the \$BANK, \$IMPORT, and \$EXPORT directives, to get a similar effect in the object module environment. There are, however, a few \$INFO groups with no object module output routines (OMOR) analog.

#### 10.9.1.1. \$INFO Group 1

Group 1 sets quarter-word and third-word sensitivity and arithmetic fault mode. This feature is no longer supported. An I flag is issued for \$INFO group 1 directives while creating object modules.

#### 10.9.1.2. \$INFO Group 11

Group 11 generates a void bank, optionally marks it as an I-bank, and can set the shared and dynamic flags. The shared attribute and dynamic attribute do not exist in the object module environment.

### 10.9.2. Collector-Defined Symbols

The Collector recognizes the following reserved symbols, and provides special information about them at collection time. Some of these features are not applicable to the object module environment and others have better ways of retrieving the information. None of these reserved names is resolved by the Linking System. If these reserved names are used, no error message is generated by MASM. Therefore, these symbols can be defined in another element.

BBJ\$	FIRSTBDI\$	LBDI\$
BDI\$	FIRSTD\$	LBDICALL\$
BDICALL\$	FIRSTI\$	LBDIREF\$
BDIREF\$	IBJ\$	MAPBDR\$
BDIRECT\$	LAST\$	MAPBDT\$
BLOCKSIZE\$	LASTD\$	MAPCALL\$
COMMN\$	LASTDR\$	MAPGOTO\$
D\$ATE	LASTI\$	SLT\$
DBJ\$	LASTIR\$	T\$IME
ENTRY\$	LASTR\$	XREF\$
FIRST\$		

### 10.9.3. Bank Switching

Multibanked programs need programmer attention to convert them to the object module environment. As previously noted, IBJ\$/DBJ\$ and BDICALL\$ and BDIREF\$ are not used. Also, retrieving the bank descriptor index (BDI) of a bank by referencing the bank name is not supported. These operations need to be rewritten, using the features described above.

### 10.9.4. Entry-Point START\$

The start address of an object module is identified by the reserved word START\$. MASM automatically generates this definition if there is a start address on the level 1 \$END image. Alternatively, a level 0 definition for START\$ can be created by the user. If an address is coded on the \$END line and START\$ is defined at level 0, the definition must be equal to the address given on the \$END line; otherwise, a D flag is issued.

# Section 11

## Symbolic Output

This section describes how to create a symbolic element from MASM source output for use as input to other processors.

### 11.1. \$SYM — Establish Symbolic Output Mode

The \$SYM directive establishes output mode for the main assembly. This allows the insertion of symbolic images produced by the MASM processor into a program file as a symbolic element. This symbolic element is suitable for input to other processors using the symbolic input/output routine (SIR\$).

The \$SYM directive must be interpreted before the beginning of the second pass of the main assembly. If an attempt is made to generate data in a symbolic output mode assembly, an I flag is produced. The \$SYM directive is ignored if it is encountered along with a \$DEF directive or if it is encountered during the second pass of the main assembly.

With a \$SYM directive in effect, the resultant output field of the processor call line (`@MASM,options source-input,resultant-output`) is used to name the symbolic output element.

When both the source input and relocatable output fields are exactly the same, the version name in the relocatable output field is shifted right by one character and a \$ sign is inserted as the first character.

**Note:** *\$SYM has the synonym \$SOR.*

### 11.2. \$OUTPUT — Output Symbolic Images

Call the \$OUTPUT directive as follows:

```
$OUTPUT,  $e_0$   $e_1, e_2, \dots, e_n$ 
```

where:

$e_0$

indicates the output options MASM uses to process symbolic images. The following output options are allowed:

P

Inserts the images  $e_1$  through  $e_n$  into the symbolic output element as though they are Fielddata images. Trailing Fielddata blanks in each image are suppressed on the next word boundary when they result in the generation of an additional word of blanks.

Q

Inserts the images  $e_1$  through  $e_n$  into the symbolic output element as though they are ASCII images. Trailing ASCII blanks in each image are suppressed on the next word boundary when they result in the generation of an additional word of blanks.

A

Performs symbolic source output on word boundaries. If the image to be output includes enough blanks to generate one or more additional words of output consisting of only blanks, these additional blanks will also be output instead of being suppressed.

$e_1, e_2, \dots, e_n$

each of these variables produces a separate symbolic image.

Any character other than P, Q, or A in the option field is ignored.

If the  $e_0$  field is blank or if only the A option is specified, the current character type of the first image,  $e_1$ , is used to indicate the character type for the source images to be output by this \$OUTPUT directive.

**Note:**  $e_0$  is not converted to a value according to the standard rules of conversion. This means its syntax is inconsistent with other MASM directives.

The P and Q options on the MASM \$OUTPUT directive are not the same as the P and Q options on the MASM processor call. They are similar only in the sense that P indicates Fielddata and Q indicates ASCII.

On the MASM processor call, these options inform the SYSLIB SIR\$ routines to convert the output symbolics into either Fielddata (P) or ASCII (Q) when necessary. The entire output symbolic is either Fielddata or ASCII.



On the MASM \$OUTPUT directive, specification of the P or Q option is only used to indicate a character type for the source images being output. No conversion is performed on any images  $e_1$  through  $e_n$  even if they are not the character type indicated by the P or Q option. However, a warning in the form of a ? flag is provided if the character type does not match. This capability can output symbolic elements containing both Fielddata and ASCII images.

Additionally, Fielddata images or nonstandard ASCII character images can be output as ASCII images just by including them as parameters on a \$OUTPUT,Q directive. The same type of process can be done with the P option.

If support for Fielddata images is disabled, a SOR routine warning message is generated and the \$OUTPUT directive with a P (Fielddata) option results in all the output images being converted to ASCII. Thus nonstandard Fielddata images are converted to ASCII as if they were standard Fielddata images. Only ASCII type output is allowed when Fielddata is disabled.

### Example

**Note:** *In this example, the spaces within quotes are significant.*

```

1.          $SYM
2.          $ASCII
3.      AI   $EQU      'Image 2  '
4.          $OUTPUT,Q  'Image 1',AI
5.          $OUTPUT,QA 'Image 1',AI
6.          $END

```

### Explanation

**Note:** *In this explanation, the symbol ^ equals an ASCII space.*

Line 1

Establishes source output mode format.

Line 2

Establishes the ASCII character set for this assembly.

Line 3

Equates AI to an ASCII character string consisting of 10 characters (with the 3 spaces after the number 2) which is 2 1/2 words of ASCII characters.

### Line 4

Inserts 2 source images into the symbolic output element. Insertions are performed on word boundaries. The first image is 2 words of ASCII characters inserted as “Image^1^”. Note the extra space to complete the word boundary. The second image is 2 1/2 words of ASCII characters but only 2 words are inserted since trailing blank words are suppressed. This image is inserted as “Image^2^”.

### Line 5

Inserts the same 2 source images into the symbolic output element as line 4. However, trailing blanks are not suppressed. The first image has no trailing blank words and is inserted the same as in line 4 as “Image^1^”. The second image is inserted as 3 words of ASCII characters, since an extra blank word is generated for the last 2 spaces in AI. This image is inserted as “Image^2^^^^”.

### Line 6

Terminates the assembly.

# Section 12

## Saving Definitions

This section describes how to save the results of processing a set of definitions in an omnibus element for subsequent use in other assemblies.

### 12.1. Definition Mode Assembly

A definition mode assembly saves the result of processing a set of definitions, so that it can be used in several assemblies. The dictionary is built by MASM as for any assembly. When the assembly is completed, the set of definitions external to the main assembly is saved in an omnibus element whose name and file are determined by field 2 of the MASM processor call statement. When the \$DEF directive is used in an assembly, the assembly is in definition mode. A definition mode assembly cannot execute any operations that generate code. The results of a definition mode assembly are retrieved by using the \$INCLUDE directive, which specifies the name of the omnibus element created. The use of definition mode is considerably faster than the other method for processing definitions, since MASM is required to read the source language only once. Definitions retrieved by the \$INCLUDE directive are in an internal format and can be placed in the dictionary faster than by scanning various directives, such as \$EQU or \$EQUF.

As an example, a set of register definitions can be loaded using the procedure definition processor (PDP)

```
@PDP,I AXR$
X1      DEF
        EQU      1
X2      EQU      2
.
.
.
R15     EQU      79
AXR$*   PROC     0,0
        END

@MASM,I PROGRAM
        AXR$
.
.
.
        END
```

## Saving Definitions

---

In this case, the definition lines must be assembled each time the AXR\$ procedure is called. The DEF directive in PDP functions in a completely different manner from DEF in MASM, as described in the PDP reference manual.

To achieve the same result using a MASM definition mode assembly, enter the following instructions:

```
@MASM,I AXR$
X1          DEF
            LEVEL      0,1,0
            EQU        1
            .
            .
            .
R15         EQU        79
            END
@MASM,I PROGRAM
            INCLUDE     'AXR$'
            .
            .
            .
            END
```

In this case, the definitions need to be assembled from source language only once. The LEVEL directive makes the given definitions external to the main assembly level. You can do this by externalizing each label defined, but that requires more coding. This is one of the most frequent uses of the \$LEVEL directive.

MASM does not resolve relocation by an external reference if the symbol is included in an element in which the external reference is defined. The Collector must resolve the relocation.

### Example

```
1.      @MASM,IS ELT1
2.
3.      DEF
4.      AZ          LEVEL      0,1,0
5.      EQU        TAG
6.      @MASM,IS ELT2
7.      INCLUDE     'ELT1'
8.      TAG*        +          2
9.      LA          A0,AZ
10     END
```

**Explanation**

Lines 1-5

A definition mode assembly in which the symbol AZ (line 4) is defined and is relocatable by the external reference TAG.

Lines 6-10

An element that has the symbol TAG defined (line 8). The statement at line 9 produces relocatable binary output that is relocatable by the external reference TAG. The Collector must satisfy this.

## 12.2. \$DEF — Establish Definition Mode

The \$DEF directive establishes definition mode for the main assembly. This directive must be interpreted before the beginning of the second pass of the main assembly. If an attempt is made to generate data in a definition mode assembly, an I flag is generated. This directive is ignored when encountered during the second pass of the main assembly.

The \$DEF directive of MASM is distinct from the DEF directive used by the procedure definition processor (PDP). Since the DEF directive should not occur inside procedures, the two never interfere with each other. The PDP DEF directive can still be used in procedures to be processed by MASM. See the PDP reference manual for details of the use of the DEF directive with the procedure definition processor.

## 12.3. \$INCLUDE — Include Definitions

The format of the \$INCLUDE directive is as follows:

```
$INCLUDE e
```

where *e* is a string in the system character set that names an element in a program file.

If the current assembly pass is generative, there is no action taken. Otherwise, *e* is assumed to be the name of an omnibus element produced by a MASM definition mode assembly (see 12.1). If found, the definitions contained in the element are added to the dictionary (always at level 1) for the present assembly. Any previous definitions for the same symbols are replaced.

### Example 1

```
$INCLUDE 'qualifier*file-name.element/version'
```

This line includes the omnibus element *element/version* from the user-defined file *qualifier\*file-name*.

### Example 2

```
$INCLUDE '.element/version'
```

This line includes omnibus element TPF\$.*element/version*.

### Example 3

```
$INCLUDE 'qualifier*file-name.element'
```

This line includes omnibus element *element* from the file *qualifier\*file-name*.

### Example 4

```
$INCLUDE 'element'
```

This line causes a search of the assembler libraries for the element (see NO TAG); if found, the element is included.

### Example 5

```
$INCLUDE 'element/'
```

This line causes a search of the assembler libraries for the first element with a version name in the file; if found, the element is included.

# Section 13

## Error and Warning Diagnostics

MASM generates warning flags (see Table 13-1) and error flags (see Table 13-2). These flags can be found in the first portion of each line; each flag can appear once or not at all on a line.

In general, the error flags mark the output element in error, which is later noted by the Collector when an absolute program is built. The diagnostic flags do not mark the relocatable binary output as being in error, although the presence of warning flags might indicate a programming problem that causes incorrect execution.

**Table 13-1. MASM Warning Flags**

Flag	Meaning
B	A base register specification may be needed for this instruction. This flag will be replaced by an A flag, if the A option is specified on the MASM processor call.
G	A \$GO directive has transferred control from one procedure to a NAME defined in a different procedure (lateral transfer of control without change of nesting level).
U	Undefined identifier used on this line.
W	Warning. Information that might be vital to this assembly is missing. However, absence of this information might not prevent the assembly from completing.
?	Improbable coding sequence or mixed modes. The results might not be what the programmer intended to generate.

**Table 13-2. MASM Error Flags**

Flag	Meaning
A	Addressability error
C	Discrepancy between location counter values of pass 1 and pass 2.
D	Redefinition of a label originally defined by implication.
E	Error in syntax, or other miscellaneous errors not included in a more specialized flag.
F	Fatal error. Indicates a system error with the file being used or the library routines (SYSLIB) being referenced. If an additional error of "ERROR using SI,SO file SIR\$ status:00000000026" appears, then the rem pack is full. Normally, only one F flag is generated and the assembly is terminated.
I	Error in directive field: Unknown directive, or attempt to increment a blocked location counter by use of the \$RES directive.
L	Level errors, such as incorrect number of \$END directives.
M	Microstring error.
Q	Missing quote terminator.
R	Relocation error - loss of relocation information due to mixed mode arithmetic, multiplication, or division of a relocatable value by a value other than one or zero.
T	Truncation of significant bits, value out of range, miscellaneous lost data.
V	Inappropriate value - integer where control information was expected.

If you use the E or F option on the MASM call line, the listing produced by MASM contains information that might help determine how errors and warnings were generated. This includes a walkback within a procedure nest. Without these options, each flag is printed only once per line. Therefore, the count of errors and warnings in the END MASM line can be greater than the number of error and warning flags in the listing. The E option prints a line each time an error is recognized. The F option prints a line each time a warning is recognized. Thus the E and F options provide more detailed diagnostic reporting.

If a user tries at level 0 to define a symbol that is relocated by an external reference, the symbol is dropped from the entry-point table and the element is marked in error.



# Appendix A

## OS 1100 Features

In an unaltered environment, MASM generates code for OS 1100 hardware architecture. This appendix describes OS 1100 features that are built into MASM.

### A.1. Instruction Mnemonic Redefinition

The M option on the processor call card allows you to redefine all OS 1100 instruction mnemonics (see 2.4). This option increases assembly time substantially because the libraries are searched for every directive not typed as a procedure.

There is a subset of the OS 1100 instruction repertoire that is always redefinable, regardless of the presence or absence of the M option. This subset contains the instructions shown in the following table:

<b>f</b>	<b>j</b>	<b>a</b>	<b>Mnemonic</b>	<b>Description</b>
05	17	01	SNZ	Store negative zero
05	17	10	INC	Increment by one
05	17	11	DEC	Decrement by one
33	02		BTT	Byte translate and test
33	05		BPD	Convert byte to packed decimal
33	06		PDB	Convert packed decimal to byte
33	07		EDIT	Edit
37	00		QB	Compress quarter-word byte to binary
37	01		BQ	Expand binary to quarter-word byte
37	02		BHQ	Compress quarter-word byte to halves
37	04		QDB	Quarter-word to double binary
37	05		DBQ	Double binary to quarter-word byte

continued

f	j	a	Mnemonic	Description
73	17	00	TS	Test and set
74	04		JK	Console selective jump

Another way to redefine instructions is to introduce the redefinition procedures through a definition mode assembly. You include the procedure definitions in the element. The procedures replace the standard instruction definitions.

## A.2. Extended and Basic Mode Operation

The following subsections describe the changes that you must be aware of when using MASM to generate the code for an extended mode OS 1100 system.

### A.2.1. Mode Directives

There are three generation modes in MASM:

- No mode  
Set by the absence of a mode directive (see A.2.2)
- Basic mode  
Set by the \$BASIC directive (see A.2.3)
- Extended mode  
Set by the \$EXTEND directive (see A.2.4)

Many instruction mnemonics are valid on both the basic mode and extended mode portions of the extended mode machine architecture. This means that these instruction mnemonics have multiple instruction formats and possibly multiple operation codes.

When you specify instruction mnemonics that do not exist on the no-mode machine architecture, you must specify a mode environment with a \$BASIC or \$EXTEND directive (see A.2.3.2 and A.2.4.2 respectively). (The no mode environment consists of only those instruction mnemonics valid on supported hardware before the introduction of extended mode machine architecture.)

Generation mode and base register environment definitions are defined at the assembly level unless generation mode is given on the \$PROC directive (see A.2.13). When you specify a particular mode environment, MASM generates and processes the instruction format for this mode environment. You should specify the directive before you specify a location counter.

**Note:** Due to different programming requirements, program either in a \$BASIC mode or \$EXTEND mode environment. Avoid mixing modes in an assembly.

## A.2.2. No Mode Operation

No mode is set by the absence of a mode directive. The no mode environment consists of only those instruction mnemonics valid on supported hardware before the introduction of extended mode machine architecture. Most of these instruction mnemonics are still valid on the basic mode portion of the extended mode machine architecture. If you use instruction mnemonics that are restricted to this set of no mode instruction mnemonics, you do not need to specify a \$BASIC or \$EXTEND directive even on the extended mode machine architecture.

When neither the \$EXTEND or \$BASIC directive is encountered, the state of the assembly remains as defined in the previous levels of MASM, before OS 1100 extended mode machine architecture. New instructions, however, can conflict with existing code.

### A.2.2.1. No Mode Syntax

In no mode, the syntax is the same as the traditional OS 1100 assembly language. The format is as follows:

$$f, j \quad a, *u, *x, j$$

or

$$f, j \quad *u, *x, j$$

### A.2.2.2. No Mode versus Basic Mode

As shown in A.2.2.1 and A.2.3.1, the no mode format and the basic mode format are not synonymous. The introduction of the b-field in extended mode affects the specification of a j-field in the operand field of an instruction statement. The result is a syntax change that must be identified by the \$BASIC directive for all instruction mnemonics for the basic mode portion of the extended mode machine architecture.

#### Example

- The instruction mnemonic AA is valid in no mode, basic mode, or extended mode. The existing mode environment determines the syntax of the instruction.
- The instruction mnemonic EX is valid in no mode, basic mode, or extended mode. The existing mode environment determines the syntax and the operation code.
- The instruction mnemonic SE is valid in no mode or basic mode. The existing mode environment determines the syntax. This instruction mnemonic is not valid in extended mode.
- The instruction mnemonics ADD1 and SUB1 are valid in basic mode or extended mode. The existing mode environment determines the syntax. The no mode syntax is not valid for these instructions because they did not exist before the extended mode machine architecture.
- The instruction mnemonic CALL is valid only in extended mode.

**Note:** *MASM does not distinguish between instruction mnemonics that are privileged, such as ADD1 and SUB1 in basic mode, and those that are not privileged. An instruction mnemonic that appears valid at assembly time can be invalid at execution time if the correct user privileges do not exist. Refer to the appropriate hardware documentation for this information.*

### A.2.3. Basic Mode Operation

Basic mode is the mode of execution on OS 1100 extended mode machine architecture systems that is compatible with no mode machine architecture systems such as the 1100/60. When you specify basic mode instruction mnemonics that do not exist on the no mode machine architecture, you must specify a mode environment with a \$BASIC directive.

#### A.2.3.1. Basic Mode Syntax

In basic mode the j-field is no longer allowed as part of the operand field. The format is as follows:

$f, j \quad a, *u, *x$

or

$f, j \quad *u, *x$

**Note:** *As an aid in writing mode-independent code, remember that a b-field after the x-field does not generate an error. Therefore, a j-field does not generate an error either, but is ignored.*

#### A.2.3.2. \$BASIC Directive

The format of this directive is as follows:

\$BASIC

or

BASIC

When this directive is interpreted, the state of the assembler is changed to reflect the following:

- Instructions have the form 6, 4, 4, 4, 2, 16.
- The basic mode instruction mnemonics available with the extended mode architecture are enabled with previous no mode instruction mnemonics.
- The new instruction syntax is enforced.

**Example**

```
1.          $BASIC
2.          LA          A3, TAG
```

**Description**

Line 1

Signals MASM to use BASIC mode format rules.

Line 2

Assumes that TAG has an offset of 032, and that the value of 10 000 03 00 0 000032 is generated for the f-field, j-field, a-field, x-field, hi-field, and u-field respectively.

## A.2.4. Extended Mode Operation

Extended mode hardware provides many capabilities that solve the addressing and bank basing limitations found on older Unisys systems. With extended mode, each user activity potentially has a six billion word address space.

Extended mode is the mode of execution on 1100/90 and 2200 systems that allows

- Sixteen user base registers
- Increased address space
- Additional machine instructions

When you specify extended mode instruction mnemonics that do not exist on the no mode machine architecture, you must specify a mode environment with a \$EXTEND directive.

### A.2.4.1. Extended Mode Syntax

In extended mode, the u-field is usually replaced by a 12-bit d-field, and a 4-bit b-field is appended to the operand field. The standard format is as follows:

*f, j*      *a, \*d, \*x, \*b*

or

*f, j*      *\*d, \*x, \*b*

### A.2.4.2. \$EXTEND Directive

The format of this directive is as follows:

```
$EXTEND[ ,  $e_0$ ]
```

or

```
EXTEND[ ,  $e_0$ ]
```

where  $e_0$  is a nonrelocatable integer expression between 0 and 1.

When this directive is interpreted, the state of the assembler is changed to reflect the following:

- Instructions have the form 6, 4, 4, 4, 2, 4, 12.
- The extended mode architecture instruction mnemonics are enabled, replacing the basic mode and no mode instruction mnemonics.
- The new instruction syntax is enforced.
- If  $e_0$  is an expression whose result is 1, the assembler will generate 18-bit operands if the instruction allows them and the x-field is zero. If the instruction allows 18-bit operands, but the x-field is non-zero, a 16-bit operand is generated. 18-bit addressing is available when either of the following is true:
  - An instruction specifies an immediate operand
  - The instruction is a jump or shift instruction

Only certain hardware supports 18-bit addressing. Refer to the processor and storage manuals for the hardware you are using.

**Note:** *The CJHE mnemonic has 2 valid extended mode forms. The EI\$ form is the default. The I\$ form is only available when 18-bit operands are allowed by setting  $e_0$  on the \$EXTEND directive. The \$MACH directive does not establish instruction forms; \$MACH only provides sets of instruction mnemonics.*

**Example**

```
1.          $EXTEND
2.          LA          A3, TAG
3.          $EXTEND, 1
4.          J           XREF
```

**Description****Line 1**

Signals MASM to use extended mode format rules.

**Line 2**

Assuming TAG has a displacement of 0032 and B5 is the active base register, then MASM generates 10 00 03 00 0 05 0032 for the f-field, j-field, a-field, x-field, hi-field, b-field, and d-field, respectively.

**Line 3**

Signals MASM to use extended mode format rules and to allow the generation of 18-bit operands for those instructions that allow it.

**Line 4**

Assuming XREF is an external reference not defined in this assembly, then since the \$EXTEND,1 is active, the u-field of this instruction will allow an 18-bit operand with 18-bit relocation.

**Note:** *18-bit addressing for jump instructions is generated, but the architecture determines whether full 18-bit addressing can be executed.*

### A.2.5. \$EXTEND Mode Block Transfer (BT)

In modes other than \$EXTEND, the standard syntax is used for the block transfer instruction. In \$EXTEND mode, the syntax of the BT instruction is as follows:

$$BT, j \quad x_d, b_d, *x_s, b_s, *disp$$

where:

$x_d$  and  $b_d$

indicate the destination registers.

$x_s$  and  $b_s$

indicate the source registers.

$disp$

indicates the displacement.

Because the displacement field is shared by the destination and source addresses, there is no automatic base register selection or calculation of displacement for the BT instruction.

#### A.2.5.1. Shift and Jump Instructions

These instructions do not use the b-field and, therefore, have the basic or NO mode I\$ form attached to them. Except for new instructions and new operation codes, they are the same in all three modes. The b-field is not allowed.

#### A.2.5.2. Immediate Operands in Extended Mode (j=16 or 17)

Instructions that normally use the extended mode I\$ form still have the EI\$ form attached, but the b-field is not parsed and the d-field is truncated to 18 bits instead of 12. The octal information column on the source listing shows the EI\$ form in use.



## **A.2.6. Instruction Generation**

This subsection describes the OS 1100 instruction generation features.

### **A.2.6.1. B-Field Overrides**

When two values that have the EI\$ form attached are merged together, the b-field of one of the values is overridden by the other. The rule is that the instruction with a b-field specified overrides the value assigned by the EQU directive that has a b-field specified.

### **A.2.6.2. Five-Bit B-Field**

If the user has specified registers B16 through B31 in the b-field, the I bit is set and the least significant four bits of the base register number are put in the 4-bit b-field. If the d-field is flagged, the T flag is generated.

### A.2.7. \$BASE Directive

The format of this directive is as follows:

```
[label]    $BASE[,e0]    e1[,e2,...,e32]
```

or

```
[label]    BASE[,e0]    e1[,e2,...,e32]
```

where:

$e_0$

is an integer expression with relocation or zero.

$e_1$  to  $e_{32}$

are nonrelocatable integer expressions between zero and 31.

Conversion is performed as needed. If a label is given, its value is a node reference consisting of two selectors. Selector 1 is the value of  $e_0$  and selector 2 is a node reference consisting of 1 to 32 selectors, each holding a base register. Selectors are numbered, starting with 1.

After interpreting this directive, MASM assumes that base register  $e_1$  points to location  $e_0$  at run time. If there is more than one base register specified,  $e_2$  is assumed to point to  $e_0+4096$ ,  $e_3$  to  $e_0+2*4096$ , etc. It is the responsibility of the program to load the base registers. The base registers declared on the \$BASE directive are referred to as a set of base registers. Many \$BASE directives can be in effect at the same time. The set of all active \$BASE directives is the base register environment definition (BRED).

When generating memory references that require the determination of a base register and displacement, MASM calculates the distance from  $e_0$  to the location and then chooses base register  $e_1$  so that the displacement is positive and is less than 4096. If no such base register exists, an addressability error (A flag) is logged and the relocatable output is marked in error. The d- and b-fields can be coded explicitly. The base register can be flagged. If flagged, the value of the b-field is used with the value of the d-field, with or without relocation, and no calculations or checking are done other than truncating b to 4 bits and d to 12. If the base register is not flagged, MASM calculates the displacement as the distance from the given register to the relocatable value. If this displacement is negative or greater than 4095 or if the base register is not active, the line is treated as though it had a flag on the b-field.

A base register is declared active by a \$BASE directive with a relocatable expression or a \$USE directive (see A.2.8). A base register is deactivated by a \$BASE directive with  $e_0$  unspecified or zero. A base register can also be deactivated at the end of the subassembly that activated it. When a base register is activated by a \$BASE directive, the base register has a location counter and offset associated with it. An active base register with a location counter and offset covers the 4K area of main storage, starting at location counter LC+offset. A base register set of  $n$  registers covers  $n*4K$  area of main storage. A base register cannot cover more than one location counter at the same time. Base register definitions are available in lower level subassemblies, unless the base register is redefined in the subassembly. These redefinitions can be canceled at the end of the subassembly. Symbolic memory references can be made only if the area is covered by an active base register. See A.2.13 for the interaction of subassemblies with base register environment definitions and generation modes.

Base registers and areas of memory can be redefined within a single subassembly. When a base register is used in more than one \$BASE directive in the same subassembly, the most recent definition is used. When the same location counter and offset are covered by more than one base register set, a base register is chosen to yield the smallest value in the d-field. An area of memory can be covered by more than one base register. However, one base register covers only one 4K area of main storage.

### Example 1

The following is an example of a simple 4K window:

1.	\$BASE, TAG	B5
2.	LA	A3, TAG+5

### Explanation 1

Line 1

Directs MASM to use register B5 as the base register for labels between TAG and TAG+4095. Calculate displacements from TAG.

Line 2

Generates 10 00 03 00 0 05 0005.

### Example 2

The following is an example of a window larger than 4K:

1.	\$BASE, TAG	B5, B4, B7
2.	LA	A3, TAG+5
3.	LA	A4, TAG+4096+5

### Explanation 2

Line 1

Directs MASM to use register B5 as the base for labels between TAG and TAG+4095, register B4 between TAG+4096 and TAG+2\*4096-1, and register B7 between TAG+2\*4096 and TAG+3\*4096-1. Displacements are calculated to be less than 4096.

Line 2

Generates 10 00 03 00 0 05 0005.

Line 3

Generates 10 00 04 00 0 04 0005.

### Example 3

The following is an example of deactivating base registers:

1.	\$BASE, TAG	B5
2.	LA	A3, TAG+5
3.	\$BASE, 0	B5
4.	LA	A3, TAG

### Explanation 3

Assume that there are no other \$BASE directives.

Line 3

MASM no longer uses register B5 for an implicit base register.

Line 4

An addressability error (A flag) is generated since there is no base register covering the specified label.

### A.2.8. \$USE Directive

The format of this directive is as follows:

```
$USE      exp
```

or

```
USE      exp
```

where *exp* is either a nonrelocatable integer value in the range 0 to 31 or an integer value with relocation.

The \$USE directive deletes the current base register environment definition and specifies the base register to be used in the b-field of every extended format OS 1100 instruction or \$EQUF. No location counter or offset is associated with the register and no displacement is calculated while the \$USE directive is in effect. The \$USE directive is effective until another \$USE, \$BASE, or \$LIT directive, with base register specification, is encountered. \$USE can be turned off at the end of a subassembly. The d-field is filled with the value given on the instruction line or zero. Relocation, if given, is maintained. In general, if MASM calculates the d-value, relocation is not retained; but if the d-value comes directly from the source line, any relocation present is included in the relocatable binary element.

If *exp* is nonrelocatable, then it specifies the base register directly. If *exp* is a relocatable expression, MASM chooses the base register that is active for the location counter and the offset given by *exp*. If no such base register exists, an addressability error (A flag) and directive-ignored error (I flag) are logged, and the relocatable binary element is marked in error.

### Example 1

The following is an example of explicit choice of base register:

1.		\$USE	B9
2.		LA	A5, TAG
3.	TAG	+	0

### Explanation 1

Line 1

Directs MASM to use register B9 for all implicit base register generations and not to calculate any displacements.

Line 2

Generates 10 00 05 00 0 11 0001, assuming TAG is at location counter offset 1.

### Example 2

The following is an example of implicit choice of base register:

1.		\$BASE, TAG	B5
2.		\$USE	TAG
3.		LA	A9, TAG+6
4	TAG	+	0

### Explanation 2

If TAG is at location counter offset 1, line 3 generates 10 00 11 00 05 0007. If line 2 is removed, line 3 generates 10 00 11 00 05 0006.

## A.2.9. Literals

The base registers covering a literal pool are given on the \$LIT directive. The \$LIT directive with base register specification is a special case of the \$BASE directive. The format for this directive is as follows:

[label]      \$LIT       $e_1[, e_2, \dots, e_n]$

A base register is selected from the set of base registers by the rules described under the \$BASE directive (see A.2.7). The base register is used for all literals without an explicit base register until the next \$LIT directive. The base register set is assumed to point to the beginning of the literal pool defined by the \$LIT directive. The value of a literal is a location counter and offset. This value can also be covered by a base register declared on a \$BASE directive. Placing a tag at the beginning of a literal pool and using that tag on a \$BASE directive has the same result as putting a base register on a \$LIT directive. Literals with the same value are duplicated only if they are under different location counters.

## A.2.10. \$BREG Function

The \$BREG function requires one parameter or no parameter. The value of  $e$  should be an integer expression if given.

The value of \$BREG( $e$ ) is the base register associated with  $e$ . If  $e$  has an extended mode form attached, the value of \$BREG( $e$ ) is the base register specified in the b-field, if any; otherwise it is -1.

If  $e$  is an integer expression with relocation, the value of \$BREG( $e$ ) is the active base register that covers  $e$ , if one exists; otherwise, it is -1.

The value of \$BREG with no parameter is the current base register specified by the \$USE directive. If a \$USE directive is not in effect, then the value of \$BREG is -1.

### Example

```
@MASM, SER WORK.BREG/EXAMPLE, TPF$.REL
MASM xxRyy

SYS$LIB$*SYS$LIB.MAXR$ FOUND FOR $INCLUDE
1.          $EXTEND
2.          $INCLUDE 'maxr$'
3.          $(0)
4.          tag $RES 4096
5.          tag2 $RES 1
6.          b7equf $EQUF tag,,,b7
7.          .
8.          +$BREG . -1, since no $USE in effect
9.          +$BREG(tag) . -1, tag not covered by base register
10.         +$BREG(b7equf) . 07, b7 from extended mode $EQUF
11.         .
12.         $USE b9 . b9 is USE base register
13.         .
14.         +$BREG
15.         +$BREG(tag) . -1, tag not covered by base register
16.         +$BREG(b7equf) . 07, b7 from extended mode $EQUF
17.         .
18.         $BASE,tag b3,b4
19.         .
20.         +$BREG . -1, no $USE in effect
21.         +$BREG(tag) . 03, tag covered by b3
22.         +$BREG(tag2) . 04, tag2 covered by b4
23.         +$BREG(b7equf) . 07, b7 from extended mode $EQUF
24.         .
25.         $END

010000
010000
00 00 00 00 0 07 0000

010001 777777777777
010002 777777777776
010003 000000000007

010004 000000000011
. 011, b9 from $USE
010005 777777777776
010006 000000000007

010007 777777777776
010010 000000000003
010011 000000000004
010012 000000000007

END MASM - LINES: 50 TIME: 3.058 STORAGE: 28955
```

### Explanation

#### Line 1

Enables extended mode instruction format.

#### Lines 4-5

Defines the labels TAG and TAG2 with values 0 and 01000 respectively. Both labels are relocatable relative to location counter 0.

#### Line 6

Defines the label B7EQUF explicitly with the \$EQUF directive. Gives the base register B7 explicitly and attaches the extended mode form.

Line 8

Generates -1 since a \$USE directive is not in effect.

Line 9

The symbol TAG has relocation relative to location counter 0, so a search is made of the active base registers for a base register that would cover the location counter and offset. Since there are no active base registers, -1 is generated.

Line 10

The symbol B7EQUF has the extended mode form attached, so the base register is extracted from the value and 07 is generated.

Line 12

Specifies the base register B9 as the \$USE base register.

Line 14

Generates the value 011 for the base register B9.

Line 15

Generates -1 for the same reasons as at line 9.

Line 16

Generates 07 for the same reasons as at line 10.

Line 17

Declares the base registers B3 and B4 active. Assumes that the base register B3 points to location TAG and B4 at TAG+4096.

Line 18

The \$USE base register was deactivated by the \$BASE directive at line 17. Thus, -1 is generated.

Lines 19-20

The symbols TAG and TAG2 have relocation relative to location counter 0, so a search is made of the active base registers for a base register that would cover the location counter and offset. The base register B3 covers the symbol TAG, and base register B4 covers the symbol TAG2, generating 3 and 4 at lines 19 and 20 respectively.

Line 21

The symbol B7EQUF still generates 07 for the same reasons as at line 10.



### A.2.11. I\$ and EI\$ Forms

MASM provides two built-in \$FORM definitions which correspond to typical OS 1100 instruction word formats. The I\$ form definition is used by the OS 1100 instructions that are valid in NO mode and \$BASIC mode. The EI\$ form definition is used by most of the OS 1100 instructions that are valid in the \$EXTEND mode. In \$EXTEND mode, however, some instructions, notably JUMP instructions, use the I\$ form instead of the EI\$ form. MASM internally provides the appropriate form instruction for the OS 1100 instruction being generated, according to the mode in effect. These forms are also used when MASM generates the results of a \$EQUF directive. These form definitions can be perceived as follows:

I\$	\$FORM	6,4,4,4,2,16	.
EI\$	\$FORM	6,4,4,4,2,4,12	.

These form definitions are available for use by any MASM programmer.

### A.2.12. \$EQUF Directive (Equate a Field)

For documentation purposes, or if repeated references are made to the u-field, x-field, j-field, and b-field of a word, you might want to have a symbol reference these fields. The \$BASIC and NO mode formats of this directive are as follows:

```
label    $EQUF    *u,*x,j
```

or

```
label    $EQUF,j    *u,*x
```

The \$EXTEND mode formats of this directive are as follows:

```
label    $EQUF    *u,*x,j,*b
```

or

```
label    $EQUF,j    *u,*x,,*b
```

where *u*, *x*, *j*, and *b* are converted to binary values. The label is assigned a binary value, with the I\$ or EI\$ form attached according to the mode in effect.

For NO mode and \$BASIC mode, the label is assigned the values of *u*, *x*, and *j* in bits 0-15, 18-21, and 26-29 respectively, with appropriate relocation items attached.

For the \$EXTEND mode, the label is assigned the values of  $u$ ,  $x$ ,  $j$ , and  $b$  in bits 0-11, 18-21, 26-29, and 12-15 respectively, with appropriate relocation items attached. If  $u$  is flagged, bit 16 is set. If  $x$  is flagged, bit 17 is set. If  $b$  is flagged, the value of the b-field is used with the value of the d-field, with or without relocation, and no calculations or checking are done.

The \$EXTEND (extended) mode format can appear in a definition mode assembly if the proper mode is set. For extended mode formats, the use of the b-field necessitates that the u-field also be specified and vice-versa. The b-field can be supplied by any \$BASE or \$USE directive in effect. If a b-field value is not desired for the label on the \$EQUF in extended mode, no \$BASE or \$USE directive should be in effect. The b-field is ignored with no flag if in \$BASIC mode. An error flag is generated if the b-field is specified in NO mode.

The \$EQUF is recognized only in the mode under which it was defined (NO mode and \$BASIC are the same for this case). \$EQUF from a different mode is treated as a binary value and the form is ignored. This can result in the generation of a T flag.

### Example 1

```
IOBUFAD  $EQUF      4,X3,H2
```

For NO mode and \$BASIC mode, the symbol IOBUFAD is associated with the values 4,3,1, and the I\$ form is attached. The result in instruction format is as follows:

```
00 01 00 03 0 000004
```

For the \$EXTEND mode, the same values are associated with the symbol IOBUFAD and the EI\$ form is attached. The result in instruction format is as follows:

```
00 01 00 03 0 00 0004
```

If the symbol IOBUFAD is used with the OS 1100 instruction LA, such as:

```
LA      AO,IOBUFAD      .
```

the result in instruction format is as follows:

```
10 01 00 03 0 000004      .  NO mode and $BASIC mode
```

or

```
10 01 00 03 0 00 0004      .  $EXTEND mode
```

If a b-field is attached or a \$BASE or \$USE directive is in effect, NO mode generates an E flag for an attached b-field and ignores the directives; \$BASIC mode ignores all indications of a b-field; and \$EXTEND mode inserts the base register value in the b-field. For the LA instruction above, assume \$USE B6 is in effect. The extended mode result is as follows:

```
10 01 00 03 0 06 0004 . $EXTEND mode
```

The same results can be arrived at by the following instruction:

```
LA,H2      A0,4,X3 .
```

### Example 2

```
IOFUNC      EQU F      ,X2,S2 .
```

For NO mode and \$BASIC mode, the symbol IOFUNC is associated with the values 3,014, and the I\$ form is attached. The result in instruction format is as follows:

```
00 14 00 02 0 000000 .
```

For \$EXTEND mode, the same values are associated with the symbol IOFUNC, and the EI\$ form is attached. The result in instruction format is as follows:

```
0014 00 02 0 00 0000 .
```

If the symbol IOFUNC is used with the OS 1100 instruction SA, such as the following:

```
SA          A2,IOFUNC+3 .
```

the result is follows:

```
00 14 02 02 0 000003 . NO mode and $BASIC mode
```

or

```
01 14 02 02 0 00 0003 . $EXTEND mode
```

The same result can be achieved by the following instruction:

```
SA,S2      A2,3,X2 .
```

The b-field is handled in this example in the same way as described in example 1. Any field not specified or provided by a \$USE or \$BASE directive is zero-filled when omitted on the \$EQUF directive. For \$EXTEND instructions, j designators of U or XU can still be used to indicate 16-bit addressing. The generated instruction can be displayed with the EI\$ form, even though all 16 bits are used as the address.

### A.2.13. \$PROC Directive

A procedure can be defined to be a \$BASIC or \$EXTEND mode procedure. The syntax to do this is as follows:

$$[label] \quad \$PROC \ [e_1, e_2, e_3] \left[ \left\{ \begin{array}{l} BASIC \\ \$BASIC \\ EXTEND[, e_5] \\ \$EXTEND[, e_5] \end{array} \right\} \right]$$

If the mode parameter is specified, then the procedure is entered in the mode specified and the base register environment definition is copied from the calling subassembly. Any changes made in mode or base register environment are defined at the subassembly level; the changes are in effect only for the duration of the procedure. If the mode parameter is not specified or is illegal, the procedure is entered with the mode and base register environment definition of the calling subassembly and any changes are returned to the calling subassembly. A procedure can thus add, delete, or change base registers in the base register environment definition of its caller.

Parameter  $e_5$  is a nonrelocatable integer expression between 0 and 1. If  $e_5$  is an expression whose result is 1, the assembler will generate 18-bit operands if the instruction allows them and if the x-field is zero. If the instruction allows it and the x-field is non-zero, a 16-bit operand is generated (see A.2.4.2). Similar to the mode to which  $e_5$  is attached, this 18-bit operand designation is in effect only for the duration of the procedure.

## A.2.14. \$FORCE and \$FORCEOFF Directives

The \$FORCE directive requires no parameters. MASM enforces output relocation so that it does not exceed the respective field size. The relocation will be enforced when this directive is active, as shown in Table A-1.

**Table A-1. \$FORCE Relocation Enforcement**

Instruction Format	Immediate Operand	X-Register Specified	Relocation Enforced
Extended (EI\$)	No	NA	12-bit
Extended (EI\$)	Yes	No	18-bit
Extended (EI\$)	Yes	Yes	16-bit
Extended block transfer	NA	NA	7-bit
Basic (I\$)	No	NA	16-bit
Basic (I\$)	Yes	No	18-bit
Basic (I\$)	Yes	Yes	16-bit

The \$FORCE directive enforces output relocation without changing the way values and forms are merged. Truncation detection at assembly time will not be altered. Therefore, truncation due to the forced relocation will not be detected until collection time.

The MASM processor call R-option shows the forced relocation settings. Forced relocation exists only for MASM generated relocatable binary elements.

The \$FORCE directive is not necessary if the values merged are generated so that the forms match. For example:

```
T          $EQUF          5
LA  A0,T+(I$ ,,,,TAG)
```

The \$FORCE directive forces relocation as indicated in Table A-1 whenever it detects a standard instruction FORM attached. Any other information passed with a form that matches a standard instruction FORM can also have its relocation forced.

Forced relocation truncation problems will not be detected until collection time. They may be detected at assembly time, when values are generated so that the forms match as described in this example.

The \$FORCEOFF directive requires no parameters. It deactivates the \$FORCE directive and returns MASM to its normal output relocation generation.

## A.2.15. \$MACH Directive

This directive selects a desired OS 1100 instruction set. The \$MACH directive has the following form:

```
$MACH[ ,  $e_0$ ]      [  $e_1$ [ ,  $e_2$  ... [ ,  $e_n$  ] ] ]
```

Each parameter is optional but, if present, should be an integer value without relocation. The \$MACH directive, with no parameters specified, disables all instruction mnemonics.

If parameter  $e_0$  is omitted or is 0, the \$MACH directive provides for the intersection of the hardware instruction sets specified in parameters  $e_1, e_2, \dots, e_n$ . If parameter  $e_0$  is 1, the \$MACH directive provides for the union of the hardware instruction sets specified in parameters  $e_1, e_2, \dots, e_n$ . Any other value of  $e_0$  is considered undefined.

The effect of \$MACH  $e_1$  is to disable all instruction mnemonics that are not valid for the hardware identified by  $e_1$ . Multiple parameters,  $e_1, e_2, \dots, e_n$ , give the ability to select a set of instructions that is valid across several machines. The effect of \$MACH,1  $e_1, e_2$  is to disable all instruction mnemonics that are not valid on  $e_1$  or  $e_2$ . A \$MACH directive remains in effect until another \$MACH directive is processed.

The MASM definition element MACH\$DEF contains the symbolic names for the values in parameters  $e_1, e_2, \dots, e_n$ . MACH\$DEF is normally found by MASM in the MASM library (which, if MASM is installed using standard installation procedures, is the system file SYS\$LIB\$\*MASM). The following table lists the defined symbolic names and their corresponding instruction set as defined in MACH\$DEF:

Symbolic Name	Instruction Set
M\$60	1100/60
M\$60EIS	1100/60 with extended instruction set
M\$70	1100/70
M\$70EIS	1100/70 with extended instruction set
M\$80	1100/80
M\$90	1100/90
M\$100	2200/100
M\$200	2200/200
M\$300	2200/300
M\$400	2200/400
M\$600	2200/600

continued

Symbolic Name	Instruction Set
M\$900	2200/900
M\$3400	2200/3400
M\$3800	2200/3800
M\$SYS11	SYSTEM 11
M\$ALL	All of the sets above

Additional architecture group definitions in MACH\$DEF are shown in the following table:

Symbolic Name	Instruction Set
M\$NO_MODE	Combined M\$80, M\$60, M\$70, M\$60EIS, M\$70EIS (architecture prior to C Series)
M\$C_SERIES	Combined M\$90, M\$100, M\$200, M\$300, M\$400, M\$600, M\$SYS11 (released C Series Architecture machines)
M\$M_SERIES	M\$900 (M Series Architecture machine)

These instruction set definitions are available at assembly time with an \$INCLUDE 'MACH\$DEF' statement, if MASM is properly installed.

#### Example 1

```
$MACH  M$60      . Disable all instruction mnemonics
                  . which are not valid on the 1100/60
```

#### Example 2

```
$MACH  M$60,M$90 . Disable all instruction mnemonics
                  . which are not valid on the 1100/60
                  . and the 1100/90
```

#### Example 3

```
$MACH,1 M$60     . Disable all instruction mnemonics
                  . which are not valid on the 1100/60
```

### Example 4

```
$MACH,1 M$60,M$90 . Disable all instruction mnemonics  
                  . which are not valid on the 1100/60  
                  . or 1100/90
```

### Example 5

```
$MACH,1 M$ALL      . This is the default instruction set  
                  . in effect at the start of an  
                  . assembly.
```

## A.2.16. \$TMACH Function

The \$TMACH function returns the currently active instruction set. The active instruction set is specified using the \$MACH directive. The format is as follows:

```
$TMACH
```

The \$TMACH function requires no parameters. It returns a binary value with bits set corresponding to the active instruction set, as shown in the following table:

Bit	Meaning
0	1100/80
1	1100/60, 1100/70
2	1100/60 with the extended instruction set, 1100/70 with the extended instruction set
3	1100/90, 2200/600
4	System 11
5	2200/100, 2200/200, 2200/300, 2200/400
6	2200/900
7	2200/3400, 2200/3800
30	Intersection of instruction sets
31	Union of instruction sets

A bit is defined for each supported hardware type, with some bits corresponding to more than one architecture since these have identical instruction sets. Bit 0 is the least significant (rightmost) bit.



# Appendix B

## Object Module Evolution

Several changes have occurred for object modules since levels 4R1, 4R1A, 4R2, and 5R1 of MASM were released. Additional features in the Linking System have resulted in the following:

- The definition of additional attributes
- Additional fields in the specifications for banks (\$BANK), definitions (\$EXPORT), and references (\$IMPORT)
- Greater emphasis on extended mode environments

In order to make use of the latest features and developments in this evolving environment, the user must have the latest system base software installed. Changes affect only those users who are assembling in an object module environment indicated by the \$OBJ directive. In the original object module environment, a simple reassembly will not be sufficient in most cases, due to the probable use of OMDEF\$. OMDEF\$ is now OM\$DEF and the keywords have been standardized.

**Note:** *Unisys supports “extended mode” MASM usage (assembler directive \$EXTEND with or without assembler directive \$OBJ) only in software written by Unisys or in interfaces written by the customer that explicitly require extended mode assembler-produced elements according to the documentation written by Unisys. In a “nonextended mode” MASM usage (absence of assembler directive \$EXTEND), Unisys does not support the generation of object modules (use of assembler directive \$OBJ) but will continue to provide full support for the generation of other provided element types which are not object modules (absence of both the \$EXTEND and \$OBJ assembler directives).*

The following subsections provide a description of the changes from OMDEF\$ to OM\$DEF, as well as the new features that are available. Required changes are minimized as much as possible. However, each system base installation should be reviewed to see if modifications to existing programs would be worthwhile by incorporating new features or capabilities.

### B.1. OM\$DEF Name Changes (Formerly OMDEF\$)

OM\$DEF has gone through a major redefinition to make its attributes, forms, attribute definitions, and internal procs more identifiable and to prevent the restriction of easily duplicated names. For the most part, the changes consist of prefixing the original name with OM\$, changing an imbedded '\$' to '\_', removing a trailing '\$', and maintaining some consistency with the Linking System command language attribute names.

#### B.1.1. Attribute Name Changes

The following tables provide the symbolic attribute names in OM\$DEF with their corresponding original symbolic attribute names in OMDEF\$. All the names in OM\$DEF begin with OM\$. Except where marked with an asterisk, the names have been changed by dropping the ending '\$', changing embedded dollar signs to underscores, and prefixing the name with OM\$.

OMDEF\$	OM\$DEF
ACTIVITY\$	OM\$ACTIVITY
ANY\$	OM\$ANY
APPLICATION\$	OM\$APPLICATION
*BANK\$ONLY	OM\$BDI
BASIC\$	OM\$BASIC
CODE\$	OM\$CODE
COMMON\$	OM\$COMMON
CONSTANT\$	OM\$CONSTANT
DATA\$	OM\$DATA
*DEBUG\$	OM\$SDD
EXECUTE\$ONLY	OM\$EXECUTE_ONLY
*EXTEND\$	OM\$EXTENDED
EXTERNAL\$	OM\$EXTERNAL
GATED\$ONLY	dropped
GENERAL\$	OM\$GENERAL
HSPU\$	OM\$HPSU
INTERNAL\$	OM\$INTERNAL
*LINKVECTOR\$	OM\$LINK_VECTOR
LVE\$	OM\$LVE
MSU\$	OM\$MSU
NONE\$	OM\$NONE
NORMAL\$	OM\$NORMAL
*OFFSET\$	OM\$BDI_OFFSET
OTHER\$	OM\$OTHER
PRIVATE\$	dropped
PRIVILEGED\$	dropped
PROGRAM\$	OM\$PROGRAM

continued

## Object Module Evolution

---

OMDEF\$	OM\$DEF
PUBLIC\$	dropped
READ\$EXECUTE	OM\$READ_EXECUTE
READ\$GATED	dropped
READ\$ONLY	OM\$READ_ONLY
READ\$WRITE	OM\$READ_WRITE
RUN\$	OM\$RUN
SEMI\$PRIVATE	dropped
STRONG\$	OM\$STRONG
*SP\$	OM\$ISP
*SP\$LOCAL	OM\$ISP_LOCAL
SYSTEM\$	OM\$SYSTEM
UNDEFINED\$	OM\$UNDEFINED
*VA\$	OM\$BDI_VA
XSU\$	OM\$XSU

### B.1.2. New Attribute Names

The following table lists the new attribute names in OM\$DEF:

Attribute	Description
OM\$CALLER_LOCAL	Bank locality attribute
OM\$LBN	Reference resolution type attribute
OM\$LBN_OFFSET	Reference resolution type attribute
OM\$LBN_VA	Reference resolution type attribute
OM\$LOOSE	Reference SCS conformance attribute and definition SCS conformance attribute
OM\$KERNEL	Bank access control attribute
OM\$SHELL	Bank access control attribute

continued

Attribute	Description
OM\$STATIC	Reference strength attribute
OM\$STRICT	Reference SCS conformance attribute and definition SCS conformance attribute
OM\$TRUSTED	Bank access control attribute
OM\$ORDINARY	Bank access control attribute
OM\$PUBLIC	Bank access control attribute

### B.1.3. \$FORM Name Changes

The following table provides the original \$FORM name in OMDEF\$ with the corresponding replacement \$FORM names in OM\$DEF:

OMDEF\$	OM\$DEF
BANK\$FORM	OM\$BANK_FORM
DEF\$FORM	OM\$DEF_FORM
REF\$FORM	OM\$REF_FORM

### B.1.4. Standard Definition Name Changes

The following tables provide the original standard definition name in OMDEF\$ with the corresponding standard definition name in OM\$DEF that replaces OMDEF\$. In the following tables, BMB refers to basic mode bank and EMB refers to extended mode bank.

Standard Bank Attribute Definitions	
OMDEF\$	OM\$DEF
CODE\$BANK	OM\$CODE_BMB
<i>none</i>	OM\$CODE_EMB
DATA\$BANK	OM\$DATA_BMB
<i>none</i>	OM\$DATA_EMB

continued

## Object Module Evolution

---

Standard Bank Attribute Definitions	
OMDEF\$	OM\$DEF
LINK\$VECTOR	OM\$LV_BMB
<i>none</i>	OM\$LV_EMB
COMMON\$BANK	OM\$COMMON_BMB
<i>none</i>	OM\$COMMON_EMB

Standard Definition Attribute Definitions	
OMDEF\$	OM\$DEF
ENYTR\$POINT	OM\$ENTRY_DEF
DATA\$DEF	OM\$DATA_DEF
CONST\$DEF	OM\$CONST_DEF

Standard Reference Attribute Definitions	
OMDEF\$	OM\$DEF
CODE\$REF	OM\$CODE_REF
DATA\$REF	OM\$DATA_REF
CODE\$LVE	OM\$CODE_LVE
DATA\$LVE	OM\$DATA_LVE
CODE\$VA	OM\$CODE_VA
DATA\$VA	OM\$DATA_VA

### B.1.5. Procedure Name Changes

The following table provides the original procedure names in OMDEF\$ and the corresponding replacement procedure names in OM\$DEF:

OMDEF\$	OM\$DEF
USE\$LV	OM\$USE_LV
CALL\$	OM\$CALL
LOAD\$BDR	OM\$LOAD_BDR

## B.2. MASM 4R2 Object Module Changes

Object modules have further evolved, resulting in slight modifications to present fields and the addition of more fields to banks, references, and definitions. Here only the changes are identified. For more information, refer to the earlier descriptions (Section 10) and the applicable Linking System documents.

### B.2.1. \$BANK (Bank Field Changes)

MASM provides an additional capability for zero-filling a bank by the  $e_0$  parameter on the \$BANK directive.

Parameter  $e_1$  of the \$BANK directive contains the coded value describing the attributes for the bank. The changes to the encoded attribute values within parameter  $e_1$  are as follows:

- For MASM-generated default banks, the default bank type attribute is no longer OM\$GENERAL. An attempt is made to determine if the bank is attached to an even or odd location counter. If even, the bank type attribute of OM\$DATA is provided; otherwise, OM\$CODE is provided. As discussed in 10.3.4, it might not be wise to use only MASM default banks.
- For the owner access and other access attributes for a bank, the previously valid values GATED\$ONLY and READ\$GATED have been dropped.
- The bank mode attribute SP\$ has been changed to OM\$ISP for the new scientific processing hardware.
- For default banks generated by MASM, the default mode attribute has been changed from OM\$BASIC to OM\$EXTENDED in the default packet. However, the resulting default is based on the mode under which the bank was first referenced. The mode is set by the \$BASIC or the \$EXTEND directives. The bank for location counter 0, however, has a default mode attribute of basic, since the bank is always initialized before the mode environment is set.
- The locality attribute has the new value OM\$CALLER\_LOCAL.
- The previous domain attribute has been replaced by the access control attribute.

Dropped Values	New Values
PRIVELEGED\$	OM\$KERNEL
PRIVATE\$	OM\$SHELL
SEMI\$PRIVATE	OM\$TRUSTED
PUBLIC\$	OM\$ORDINARY OM\$PUBLIC

- The storage attribute includes the value OM\$ISP\_LOCAL for the new scientific processing hardware.
- For default banks generated by MASM, the default storage attribute has been changed from OM\$XSU to OM\$MSU. This was done because OM\$XSU always used high-performance storage units first, which is not generally desirable.
- An additional field has been added to specify the Integrated Scientific Processor (ISP) local store priority. It is a 6-bit unsigned integer attribute.

No changes have been introduced for parameters  $e_2$  through  $e_5$  on the \$BANK directive.

### B.2.2. \$IMPORT (Reference Field Changes and Extension)

Parameter  $e_1$  is a coded value containing the reference attributes. The following changes have been made in the reference attribute fields:

- The reference type attribute field contains the new attribute value of OM\$SDD.
- The reference strength attribute field contains the new attribute value of OM\$STATIC.
- The resolution type attribute value names have been renamed from BANK\$ONLY, OFFSET\$, and VA\$ to OM\$BDI, OM\$BDI\_OFFSET, and OM\$BDI\_VA.
- The resolution type attribute field contains the following new attribute values: OM\$LBN, OM\$LBN\_OFFSET, and OM\$LBN\_VA.
- A new field, the reference storage attribute field, has been added. It has the following possible values: OM\$HPSU, OM\$ISP\_LOCAL, OM\$MSU, OM\$UNDEFINED, and OM\$XSU.
- A second new field, the standard calling sequence (SCS) conformance attribute, has been added. It has the following possible values: OM\$LOOSE, OM\$NONE, and OM\$STRICT.

If \$IMPORT is used to apply attributes to a local defined value (that is, location counter + offset) that is determined by parameter  $e_3$ , then either parameter  $e_2$  must not specify a library code name or  $e_2$  must be a null string.

The \$IMPORT directive has been expanded to apply attributes to external references and location counter references. In addition to its previous implementation, parameter



$e_3$  on the \$IMPORT directive can be an integer value with simple relocation. When parameter  $e_3$  is an integer value, the resultant reference value has the absolute part and relocation of the integer value by either LC or XREF. If relocation of the integer value contains relocation attributes or a library search chain, these are overridden by parameters  $e_1$  and  $e_2$ , if provided.

### B.2.3. \$EXPORT (Definition Field Changes)

Parameter  $e_2$  is a coded value containing the definition attributes. Attribute fields for definition type and definition visibility have not changed. The SCS conformance attribute field has been added with the following possible values: OM\$LOOSE, OM\$NONE, and OM\$STRICT.

There are no changes in parameters  $e_1$  and  $e_3$ .

### B.3. MASM 5R1 Object Module Changes

Changes implemented since MASM 4R2 release are listed in this subsection. For more information, refer to Section 10 and the applicable System Base 3R1 software documents.

#### B.3.1. \$BANK (Bank Field Changes)

Parameter  $e_1$ , which contains the encoded attribute values, has the following changes:

- The bank merge order attribute field has been added immediately following the ISP local store priority attribute field. Possible values are OM\$UNDEFINED (default), OM\$FIRST, OM\$LAST, and OM\$NONE.
- The access control (ring number) attribute for the standard bank declarations OM\$LV\_EMB, OM\$LV\_BMB, OM\$DATA\_EMB, and OM\$DATA\_BMB has been changed from OM\$PUBLIC to OM\$ACTIVITY. This matches the UCS standard settings.
- The owner access attribute for the standard bank declarations OM\$DATA\_BMB, OM\$LV\_BMB, and OM\$COMMON\_BMB has been changed from OM\$READ\_WRITE to OM\$GENERAL. This was necessary to acquire the execute attribute when the Exec loads ZOOMs (zero overhead object modules). Existing user-generated basic mode banks might need to be modified to acquire the execute attribute also.

Parameters  $e_2$  through  $e_4$  are not changed.

Parameter  $e_5$  has been added, containing additional encoded attribute values as follows:

- Common block-aligned: OM\$CB\_ALIGNED
- Already zero-filled: OM\$ZEROED
- Segmented: OM\$SEGMENTED
- New paging status field: OM\$PAGED, OM\$PAGE\_LOCKED, and OM\$UNDEFINED (default)
- New hash checking level attribute provided for common banks: OM\$HASH1, OM\$HASH2, OM\$HASH3, OM\$HASH4, OM\$HASH5, and OM\$UNDEFINED (default)

OM\$ATTR\_FORM provides aid in building composite values of these attributes.

The default bank name in parameter  $e_6$  has been changed from MASM\$lc to *output-element-name\$lc*.

Parameter  $e_7$  has been added to specify the location counter of the related symbolic debugging dictionary (SDD) bank.

### **B.3.2. \$IMPORT (Reference Field Changes)**

Parameter  $e_1$ , containing the encoded attribute values, now has the reference resolution type attributes OM\$ENTRY\_VAR and OM\$OM\_ID added.

### **B.3.3. \$EXPORT (Definition Field Changes)**

Parameter  $e_2$ , containing the encoded attribute values, has the following changes:

- New definition type attribute OM\$FIXED\_GATE added
- New definition address type attribute field provided immediately following the SCS attribute field. Possible values are OM\$EIGHT\_LVL\_VA, OM\$FOUR\_LVL\_VA, OM\$TWO\_LVL\_VA, and OM\$UNDEFINED (default).

### **B.3.4. Miscellaneous Changes**

- The MASM definition element OM\$DEF is released with MASM and placed in SYS\$LIB\$\*MASM during an installation. It is used with object module assemblies. Previously, OM\$DEF was released with SYSLIB. For compatibility, SYSLIB retains the version of OM\$DEF as released with SYSLIB 75R3.
- The entry point OM\$HV\_CALL has been added to the procedure OM\$CALL in OM\$DEF.

### B.4. MASM 6R1 Object Module Changes

Changes implemented since the MASM level 5R1 release are listed in this subsection. For more information, refer to Section 10 and the applicable System Base 4R1 software documents.

#### B.4.1. \$BANK (Bank Field Changes)

Parameters  $e_1$  through  $e_4$  are not changed.

Parameter  $e_5$ , which contains additional encoded attribute values, now has the flat address attribute field added, immediately preceding the common block aligned attribute field. Possible values are OM\$UNDEFINED (default) or OM\$FLAT.

Parameters  $e_6$  and  $e_7$  are not changed.

#### B.4.2. \$IMPORT (Reference Field Changes)

Parameter  $e_1$ , which contains the encoded attribute values, includes the new reference resolution types OM\$ELT\_INFO and OM\$EP\_VA. Additionally, parameter  $e_1$  has the new standard reference attribute definition OM\$ENTRY\_VA.

#### B.4.3. \$EXPORT (Definition Field Changes)

Parameter  $e_2$ , which contains the encoded attribute values, has the new standard attribute definitions OM\$FG\_DEF and OM\$CBEP\_DEF. Additionally, parameter  $e_2$  includes the new visibility attribute OM\$SS\_EXTERNAL (Please check to see if this attribute is valid for the Linking System you are using).

# Appendix C

## Incompatibilities with ASM

This appendix explains in detail the few incompatibilities between the Series 1100 Assembler (ASM) and MASM and indicates how they can best be resolved.

### C.1. Instructions Generated for Overlap Registers

When the mnemonics L, S, A, and AN are used for operations on the overlap registers A0, A1, A2, and A3; MASM generates the operation codes for LA, SA, AA, and ANA, whereas ASM generated the operation codes for LX, SX, AX, and ANX. Virtually no significant impact is anticipated from this change.

### C.2. Current Location Counter Number

In ASM, the capability of determining the active location counter number was provided by the ampersand symbol (&) that had the desired value. This is an illegal symbol in MASM. However, this capability is available in MASM through the built-in function \$LCN. Therefore, when converting an ASM program to MASM, all uses of the ampersand should be changed to reference \$LCN.

### C.3. Location Counter Local to a Procedure

ASM allowed you to specify that code for a particular procedure was generated under a specific location counter, and the previously active location counter was saved and restored after the procedure was fully interpreted. This was done by the following syntax

```
$ (n),p      PROC      ...
```

where the location counter specification on the procedure line indicated a temporary change in location counter. This capability is available in MASM, but not with this syntax. This syntax would merely make an ordinary location counter change, just as if the \$(n) had been on the line preceding the procedure line.

In MASM, there are several alternative ways to achieve the preceding result. The value of the location counter at entry to the procedure can be saved (after determining it through \$LCN) and restored. The location counter can be reset to the value of \$ILCN. If the procedure has multiple exits and it is undesirable to place location counter restoration code at each exit, the third field of the \$PROC directive can be coded with the local location counter number.

### C.4. INFO Group 2

Although the \$INFO directive group 2 is defined for MASM with the same function as it is for ASM, the format of the directive is changed. To achieve the same effect, the following ASM line:

```
label          INFO          2 e
```

should be converted to the following MASM line:

```
$INFO          2 'label',e
```

### C.5. NEG Directive

The ASM NEG directive is not available in MASM. The design of the MASM NEG directive provides greater flexibility than can be obtained by a single expression, as is done with the ASM NEG directive.

### C.6. SETMIN Directive

MASM does not contain the ASM SETMIN directive. The effect of SETMIN can be obtained by use of the group 3 MASM \$INFO directive. A procedure can be written and included in SYS\$\*RLIB\$ to ease conversion programs containing this directive.

### C.7. TYPE Directive

MASM does not contain the ASM TYPE directive. The effect of TYPE can be obtained by using the group 1 MASM \$INFO directive. A procedure can be written and included in SYS\$\*RLIB\$ to ease conversion programs containing this directive.

### C.8. Arithmetic — 36 Versus 73 Bits

MASM arithmetic is performed to the high precision of 72 bits plus sign, whereas ASM arithmetic is 35 bits plus sign. In cases where overflow is a possibility, this means that the results generated by two assemblers can differ. This type coding is often encountered in assembly time generation of hash code values, where, assuming that the object time hash algorithm produces identical results, an arithmetic overflow while treating character strings as binary numbers is of no consequence.

There are two ways to convert an ASM program of the above type to MASM. You can alter the object-time hash algorithm to correspond to the MASM algorithm or force MASM to produce the same results as ASM. The latter can be done with the following code sequence, assuming you want the ASM sum of the values of A and B:

```

M      EQU      1* / 36 - 1
SUM    EQU      M** ( ( M** A ) - ( M** ( - B ) ) )

```

If this sequence is too cumbersome to write everywhere it is needed, write a FUNC to make the code more compact and readable.

Use similar techniques for multiplication and division, as well as floating-point arithmetic, even though these operations are not needed frequently.

## C.9. Global Relational Operators

The MASM expression  $A > B > C$  is computed as  $(A > B) ** (B > C)$ , whereas ASM would compute it as  $(A > B) > C$ . The ASM form is not as meaningful as the MASM form, so this difference in interpretation is insignificant.

## C.10. Improved Error Detection

MASM detects many more errors than ASM. The programmer converting an ASM program to MASM finds that MASM generates a number of error flags that were not present in an ASM assembly. This includes both the new error flag types of MASM as well as error flags common to both. These errors are found most often in procedures and complicated address manipulation schemes. The programmer must determine the cause of each individual error flag and eliminate them one by one.

## C.11. Greater Permissiveness of MASM

In some cases, MASM permits the programmer to write code that would generate errors under ASM. A primary example of this is the redefinition of labels. MASM permits the redefinition of all labels other than those defined by implication (that is, written as the label field of a code generating a directive or procedure). Therefore, if a label is needed to hold a value assigned by \$EQU and the label is to be given different values, in MASM an ordinary label can be used. ASM programmers were forced to use a subscripted label, thereby introducing meaningless additional characters into the coding and impairing maintainability. This is no longer necessary, and the programmer converting to MASM might want to examine the program to simplify code.

## C.12. Iteration Variables

The iteration variable for a \$DO or \$REPEAT directive with a zero repeat count retains its previous value. Under ASM, the value was set to zero.

### C.13. Propagation of Asterisk Flag

MASM does not propagate the asterisk flag as part of an expression; the flag must be reestablished where desired. This avoids unwanted generation of indirect addressing. The following sequence generates 1 in ASM, but generates 0 in MASM:

```
C1*      PROC
          +          C1(1,*1)
          END
C2*      PROC
          C1          C2(1,1)
          END
          C2          *2
```

To achieve the same results in MASM, the body of C2 should be replaced by the following line:

```
C1      [C2(1,*1)->'*!']C2(1,1)
```

### C.14. Forms Shorter Than 36 Bits

MASM right-justifies all forms shorter than 36 bits, where they were left-justified by ASM. This can affect lines referencing the implicit form of the \$GEN directive if the number of fields is not a divisor of 36. An error flag also marks these lines.

### C.15. ASM\$PF vs. MASM\$PF1

The file name ASM\$PF, used by ASM as part of its library search sequence, has been replaced in MASM by the file name MASM\$PF1. Only the file names have changed. Their operations remain the same, although MASM has a larger library search sequence.

### C.16. Forward References to \$PROC or \$FORM

Procedures and forms should be defined before they are used. ASM and MASM levels occasionally allow forward references to \$PROC or \$FORM that generated only one word. The C and D flags can be cleared up by moving \$PROC or \$FORM to a position that precedes its use.

### C.17. Directives ON, OFF, and FIELDATA

To facilitate conversion, MASM recognizes the ASM directives ON, OFF, and FIELDATA. However, they should be changed to the MASM synonymous directives \$IF, \$ENDF, and \$FDATA.



# Appendix D

## Restrictions, Operational Considerations and Compatibility

This appendix discusses current MASM restrictions, operational considerations, and compatibility issues.

### D.1. Restrictions

Restrictions are temporary limitations for a software product. Restrictions will be removed in a subsequent release of the product.

There are no restrictions for this level of MASM.

### D.2. Operational Considerations

Operational considerations are permanent and are listed here to promote the effective use of a software product.

- MASM does not support recursive (circular) data structures.
- MASM memory expansion cannot exceed 262K.

### D.3. Compatibility

MASM compatibility considerations are discussed in the following subsections.

#### D.3.1. MASM System Type Differences

Conflicts might occur between user-defined symbols and new machine instructions, which are added to the valid instruction repertoire when new systems are released. These conflicts are user dependent.

Occasionally, product releases and hardware releases do not occur concurrently. This might result in MASM containing instruction mnemonics for unreleased systems, or not containing instruction mnemonics for released systems. The latter is resolved by moving up to a new MASM release when the release is available.

Machine-specific instruction mnemonics are no longer recognized for the 1100/20, 1100/40, and earlier systems.

### D.3.2. Compatability with Previous Release Levels

The following are MASM compatibility issues:

- NOP instruction

MASM level 4R1 and lower generate a no-operation (NOP) instruction and produce an I flag, for an illegal or undefined directive.

In MASM level 4R2, only the I flag is produced, since this was considered sufficient to indicate the error.

MASM level 5R1 produces the I flag and also generates the NOP instruction.

- Cross-reference listing

The MASM cross-reference listing has been enhanced to provide additional information. Emphasis is now on those symbols processed at assembly time, rather than the symbols found in an assembly listing. Thus, some symbols previously not listed are now listed, and other symbols that were previously listed are not listed now.

- Definition mode assemblies

MASM level 3R1 and lower cannot read definition mode assemblies generated by MASM level 4R1 and higher.

MASM levels 4R1, 4R1A, and lower cannot read definition mode assemblies generated by MASM level 4R2 and higher.

Compatibility with previous levels of MASM is retained, so that higher levels of MASM can still process definition mode assemblies generated by a lower MASM level. However, for performance reasons, old definition mode assemblies should be reassembled with the new level of MASM being used.

Beginning with level 5R1, MASM searches the MASM product file (for example, SYS\$LIB\$\*MASM) for omnibus elements, created using MASM definition mode assemblies.

The MASM definition element OM\$DEF is released with MASM and must be installed in the MASM product file.

- Relocatable elements

MASM levels 6R1 and higher generate relocatable elements using the enhanced relocatable binary format. Only Collector level 33R1 and higher will process this enhanced relocatable binary format. If it is necessary to use a lower version of the Collector, all relocatable elements must be reassembled with a lower level of MASM.

# Glossary

## A

### **A register**

The register used for arithmetic.

### **A-X register**

A register that can be either an X or an A register.

### **abort**

To terminate a program abnormally.

### **absolute**

An executable element created by the Collector. It can be saved and re-executed many times.

### **address tree**

A model used to describe the address space of the 1100/50, 1100/90 or 2200/200 systems. The model is shaped like a branching tree and has four levels, called system, application, program, and activity. At any of these levels, other banks at the same level or at lower levels are visible.

Each level represents one portion of the address space. The levels provide the operating system software the ability to control access to shared information (banks) and to isolate unrelated programs from one another.

By assigning a level (Collector directive LEVEL), a program can specify at what level the information can be shared. *See* level.

## B

### **B register**

In old symbolics, B registers were occasionally used instead of X registers as index registers. *See* base registers.

### **bank**

A data structure of computer hardware representing an area of storage consisting of a set of contiguous words. Banks can vary in length. There are both upper and lower address bounds associated with each bank.

## Glossary

---

### **base register**

One of sixteen registers used to contain the BDI and base address of active program banks in extended mode. *See* B register.

### **basic mode**

The traditional OS 1100 program execution mode. It supports a two-level address tree and allows for four active program banks at a time. This is the only mode of execution on the 1100/60, 1100/70, and 1100/80.

### **BDI**

*Abbreviation for* bank descriptor index.

### **BRED**

*Abbreviation for* base register environment definition.

## **C**

### **Collector**

The OS 1100 system processor that combines or collects relocatable elements generated by MASM or the language processors with relocatable library elements to form an executable (absolute) element.

### **CTS**

*Abbreviation for* Conversational Time Sharing.

## **D**

### **directive**

A mnemonic instruction to the assembler, as opposed to an instruction mnemonic. Directives are carried out at assembly time, whereas instructions are carried out at program execution.

## **E**

### **element**

A named grouping of information manipulated as a unit. There are five basic types of elements: absolute, object module, omnibus, relocatable, and symbolic.

### **extended mode**

A program execution mode available on some OS 1100 hardware. It supports a four-level address tree and allows for up to sixteen active user program banks at one time.

## **H**

### **HPSU**

*Abbreviation for* high-performance storage unit.

### I

**index register**

A register used in hardware address modification. *Also called X register.*

**INFO**

A directive accepted by MASM. The INFO directive allows the user to code information in an assembly program that can direct the collection of the assembled output element.

**IPF 1100**

*Abbreviation for Interactive Processing Facility.*

**ISP**

*Abbreviation for Interactive Scientific Processor.*

### L

**language processor**

A processor whose principal functions include compiling, assembling, translating, interpreting, and related operations for a specific programming language (such as COBOL or FORTRAN).

**LBN**

*Abbreviation for logical bank number.*

**level**

A portion of address space that defines sharing among banks. There are two levels in basic mode and four in extended mode. *See address tree.*

**library**

A program file or files containing elements that are searched during assembly, so that they can be automatically included with the user program.

**linking**

The process of merging a set of object modules into a single object module. This primarily involves uniting separate routines into a single program, translating symbolic references to addresses, and adjusting addresses as necessary. Linking can be static or dynamic. This is analogous to collecting a program that is composed of relocatable elements.

**location counter**

An internal MASM variable that holds the next available location in any one of a set of numbered blocks that can be known to the program. Usually the block is just called the location counter. This the smallest unit of storage that can be manipulated by the Collector or the Linking System. In an object module, each location counter can become a separate bank.

### M

#### MAP

*synonym for the Collector. The Collector is invoked by @MAP.*

#### MASM

The OS 1100 Meta-Assembler, the language processor used to process the OS 1100 assembly language. It converts mnemonic and symbolic machine code into machine language.

#### MSU

*Abbreviation for main storage unit.*

### O

#### object module

A type of element in a program file that contains banks, bank control information, diagnostic information, and sufficient information to allow the object module to be statically or dynamically linked. Object modules are created by some language processors. MASM creates an object module by using the \$OBJ directive.

#### OMOR

*Abbreviation for object module output routine.*

#### operand

One of the elemental pieces of data or one of the registers used by an instruction.

#### operator

A symbol for one of the arithmetic, logical, or relational operations used in an expression. MASM executes these at assembly time; it does not assemble operators to executable code.

### P

#### PADS

*Abbreviation for programmers advanced debugging system.*

#### PDP

*Abbreviation for procedure definition processor.*

#### PLIBT

*Abbreviation for procedure library search table.*

### R

**R register**

A set of 15 additional hardware registers available to use. Certain registers are dedicated for some instructions.

**relocatable**

An element containing a program part in relocatable binary format, suitable for input to the Collector. Relocatable elements can be produced by MASM, language processors, and the Collector itself.

**relocatable output routine**

A routine called by language processors and assemblers to produce relocatable elements.

**ROR**

*See* relocatable output routine.

### S

**SCS**

*Abbreviation for* standard calling sequence.

**SDD**

*Abbreviation for* symbolic debugging dictionary.

**SIR\$**

Symbolic input/output routine.

**symbolic**

An element that contains program code, text, or data in a character representation. The most common use of symbolic elements is as source language to be input to a language or system processor.

**SYSLIB**

The System Service Routines Library. SYSLIB is a collection of service routines.

**system processor**

A processor whose principal functions are as a specialized system service or utility, not as a compiler or a language processor.

### X

#### **X register**

*See* index register.

### Z

#### **ZOOM**

*Abbreviation for* zero overhead object module.



# Bibliography

*OS 1100 Exec System Software Executive Requests Programming Reference Manual* (7830 7899). Unisys Corporation.

*OS 1100 Collector Programming Reference Manual* (7830 9887). Unisys Corporation.

*OS 1100 Linking System Programming Guide* (7831 0521). Previously titled NPE Linking System Programming Guide. Unisys Corporation.

*OS 1100 Linking System Programming Reference Manual* (7831 0505). Previously titled NPE Linking System Programming Reference Manual. Unisys Corporation.

*OS 1100 Procedure Definition Processor (PDP) Operations Reference Manual* (UP-10070). Unisys Corporation.

*1100/60 Systems Processor and Storage, Programmer Reference* (UP-9084). Unisys Corporation.

*1100/70 Systems Processor and Storage, Programming Reference Manual* (UP-9652). Previously titled 1100/70 Systems Processor and Storage Hardware Programmer Reference. Unisys Corporation.

*1100/80 Systems 4x4 Capability Processor and Storage Programming Reference Manual* (UP-8604). Unisys Corporation.

*1100/90 Systems Processor and Storage, Programming Reference Manual* (UP-9667). Unisys Corporation.

*2200/200 System Processor Complex Programming Reference Manual* (UP-11275). Unisys Corporation.

*2200/400 System Central Electronics Complex Programming Reference Manual* (UP-12452). Unisys Corporation.

*2200/600 System Processor and Storage Programming Reference Manual Volume 2, Instruction Repertoire* (UP-13397). Unisys Corporation.



# Index

## Symbols

- `$(e)` function, 5-6
- `$ANDF` directive, 6-3
- `$AP` function, 4-4
- `$ASCII` directive, description of, 4-10
- `$BA` function
  - description of, 4-4
  - selector summary (table), 4-4
  - with flag attribute, 4-32
- `$BANK` directive
  - bank attributes, 10-2
  - bank name, 10-12
  - default values, 10-27
  - description of, 10-9
  - example, 10-13
  - field changes (4R2), B-6
  - field changes (5R1), B-8
  - field changes (6R1), B-9
  - library definition element, 10-27
  - link vectors, 10-22
- `$BASE` directive, A-8
- `$BASIC` directive
  - description of, A-4
  - with `$EQUF`, A-14
- `$BREG` function, A-12
- `$CAS` function, 4-13
- `$CB` function, 4-15
- `$CD` function, 4-14
- `$CFS` function, 4-14
- `$CHAR` directive
  - description of, 4-10
  - translation table, 4-10
  - with `$ASCII`, 4-10
  - with `$FDATA`, 4-10
- `$CROSSREF` directive, 3-3
- `$CS` function, 4-14
- `$DATE` function, 8-1
- `$DEF` directive
  - description of, 12-2
  - in an assembly, 12-1
- `$DELETE` directive
  - description of, 7-2

# Index

---

- in library search, 2-21
  - with node reference, 4-18
- \$DISPLAY directive
  - description of, 3-1
  - with \$GP, 6-20
  - with \$SSS, 4-17
  - with flag attribute, 4-32
- \$DO directive
  - description of, 6-4
  - literal operand, 4-6
- \$EJECT directive, 3-3
- \$ELSE directive, 6-1
- \$ELSF directive, 6-2
- \$END directive
  - description of, 6-10
  - line level, 6-25
- \$ENDD directive, 6-5
- \$ENDF directive
  - description of, 6-2
  - with ASM, C-4
- \$ENDI directive, 6-6
- \$ENDR directive
  - description of, 6-5
  - line level, 6-25
- \$EQU directive
  - description of, 4-1
  - for control information, 4-24
- \$EQUF directive
  - b-field override, A-8
  - description of, A-14
  - with integer values, 4-2
- \$EXPORT directive
  - defaults, 10-27
  - definitions, 10-18
  - description of, 10-20
  - example, 10-21
  - field changes (4R2), B-8
  - field changes (5R1), B-9
  - field changes (6R1), B-10
  - library definition element, 10-27
  - with externalized labels, 2-16
- \$EXTEND directive
  - description of, A-5
  - with \$EQUF, A-15
- \$FDATA directive, 4-10, C-4
- \$FN function, 6-22
- \$FORCE directive, A-17
- \$FORCEOFF directive, A-17
- \$FORM directive
  - description of, 5-4
  - name changes, B-4
  - with integer values, 4-2

- \$FP function
  - description of, 6-20
  - use of, 6-20
- \$FUNC directive
  - description of, 6-13
  - line level, 6-25
- \$GEN directive
  - description of, 5-3
  - implicit call, 4-7, 5-1
- \$GFORM directive, 5-5
- \$GO directive
  - cross-reference rule, 2-7
  - description of, 6-6
  - use of, 6-15
- \$GP function
  - description of, 6-20
  - use of, 6-20
- \$HASH function, 7-2
- \$HDG directive, 3-4
- \$HEX directive, 3-3
- \$IBITS function, 4-32, 4-36
- \$IC function, 7-3
- \$IF directive
  - cross-reference rule, 2-7
  - description of, 6-1
  - literal operand, 4-6
  - with ASM, C-4
- \$ILCN function, 5-7
- \$IMPORT directive
  - defaults, 10-27
  - description of, 10-16
  - example, 10-18
  - field changes (4R2), B-7
  - field changes (5R1), B-9
  - field changes (6R1), B-10
  - library definition element, 10-27
  - references, 10-13
- \$INCLUDE directive
  - cross-reference rule, 2-7
  - description of, 12-3
- \$INFO directive
  - compatibility issues, 10-29
  - description of, 9-1
  - group 1, 9-1, 10-29, C-2
  - group 10, 9-5
  - group 11, 9-5, 10-29
  - group 12, 9-6
  - group 2, 9-2, C-2
  - group 3, 9-3, C-2
  - group 4, 9-3
  - group 5, 9-4
  - group 6, 9-4

# Index

---

- group 7, 9-4
- group 8, 9-5
- group 9, 9-5
- restrictions, 9-6
- \$INSERT directive, 6-7
- \$L0 function, 4-22
- \$L1 function, 4-22
- \$LCB function, 5-7
- \$LCFV function, 5-8
- \$LCN function, 5-7
- \$LCV function, 5-7
- \$LEV function, 7-4
- \$LEVEL directive
  - description of, 7-4
  - in definition mode assembly, 12-2
- \$LF function, with waiting label, 6-14
- \$LINES function, 8-2
- \$LIST directive, 3-1
- \$LIT directive
  - description of, 5-8
  - with base registers, A-12
- \$LP function
  - description of, 6-20
  - use of, 6-20
- \$MACH directive, A-18
- \$NAME directive
  - description of, 6-7
  - use of, 6-15
- \$NEG directive, 5-9
- \$NIL directive, 6-6
- \$NODE function, 4-22
- \$NS function, 4-23
- \$OBJ directive
  - description of, 10-1
  - UCS, B-1
  - with externalized labels, 2-16
- \$OCTAL directive, 3-3
- \$OUTPUT directive, 11-1
- \$PAR function, 8-2
- \$PROC directive
  - description of, 6-9
  - in conditional expression, 4-31
  - line level, 6-25
  - with modes, A-17
- \$REL directive, 9-1
- \$REPEAT directive
  - description of, 6-5
  - line level, 6-25

- \$RES directive, 5-6
- \$SL function, 4-16
- \$SN function, 4-23
- \$SR function, 4-16
- \$SS function, 4-16
- \$SSS function, 4-17
- \$SYM directive, 11-1
- \$TBIN function (table), 4-35
- \$TCON function (table), 4-35
- \$TDAT function (table), 4-35
- \$TDIR function (table), 4-35
- \$TFLT function (table), 4-36
- \$TFNM function (table), 4-36
- \$TFUN function (table), 4-36
- \$TINM function (table), 4-36
- \$TMACH function, A-20
- \$TMODES function, 8-2
- \$TNAM function (table), 4-36
- \$TNOD function (table), 4-36
- \$TPNM function (table), 4-36
- \$TSTR function (table), 4-36
- \$TVAL function (table), 4-36
- \$TYPE function, 4-35
- \$UNLIST directive, 3-1
- \$USE directive
  - description of, A-10
  - with base register, A-9
- \$WRD directive, for precision, 4-30
- \$XLEV function, 7-5
- @ENDX control statement, 2-4
- @USE statement, in search order, 2-22

## A

- absolute element, replacement for, 10-1
- absolute part (\$AP), 4-4
- access control
  - 5R1 changes, B-8
  - bank attribute, 10-6
- access permission, bank attribute, 10-3
- address
  - 5R1 changes, B-9
  - bank attribute, 10-8
- address of
  - data, 4-3
  - destination (BT), A-7
  - indirect, C-3
  - program start, 9-4
  - source (BT), A-7

# Index

---

- address type, definition attribute, 10-20
- alignment
  - bank attribute, 10-7
  - common block attribute, 10-9
- ampersand (*See* symbol, ampersand)
- AND operator, 4-26
- apostrophe character, 4-8
- arithmetic
  - fault mode, 10-29
  - mixed-mode, 4-24
  - operators, 4-25
  - precision, 4-2, 4-24, C-2
- ASCII
  - conversion function, 4-13
  - input, 1-1
  - string limits, 4-9
  - with \$CHAR, 4-10, 4-11
  - with \$DATE, 8-1
- ASM vs MASM (*See* compatibility issues)
- ASM\$PF
  - compatibility issues, C-4
  - in library search, 2-21
  - in search order, 2-22
- assembly
  - (*See* subassembly)
  - definition mode, 7-4, 12-1, D-2
  - storage size, 2-4
  - time required, A-1
- assembly passes (*See* passes of assembly)
- assembly-time
  - controls, 6-1
  - expression values, 4-24
- asterisk
  - arithmetic operator, 4-25
  - dictionary control item, 2-14, 2-16
  - externalized symbols, 6-13
  - flag, C-3
  - flag attribute, 4-3
  - in conditional expression, 4-31, 4-32
  - in operand subfield, 2-18
  - microstring, 6-24
  - multiplication, 4-5
  - unary (literal item), 4-6
  - unary (set flag), 4-3, 4-32
- attribute
  - flag, 4-3, 4-32
  - integer, 4-4
  - name changes, B-2
  - new names, B-3
  - of bank, 10-2
  - of definition, 10-19
  - of object module, 10-1



of reference, 10-13  
OM\$DEF in UCS, B-1

## B

### bank

- attributes of, 10-2
- basic, 10-12
- code, 10-12
- data, 10-12
- declarations in OM\$DEF, 10-27
- default attribute values, 10-12
- descriptor index (BDI), 10-9, 10-30
- example, 10-13
- extended mode, 10-12
- general purpose, 10-12
- group, 10-2
- link vector, 10-13, 10-22
- location counter, 10-2
- mode (4R2), B-6
- name with object modules, 10-12
- switching, 10-30
- type definitions in OM\$DEF, 10-27
- void, 9-5

### bank attribute

- \$BANK directive, 10-9
- access control, 10-6
- access control (4R2), B-7
- access control (5R1), B-8
- access permission, 10-3
- access permission (4R2), B-6
- access permission (5R1), B-8
- address, 10-8
- alignment, 10-7, 10-9
- bank type, 10-2
- common block, 10-9
- common block-aligned (5R1), B-8
- default values, 10-12, 10-27
- defining, 10-9
- definitions in OM\$DEF, 10-27
- hash checking level, 10-9
- hash checking level (5R1), B-9
- initialization, 10-9
- ISP local priority, 10-7
- locality, 10-4
- locality (4R2), B-6
- merge order, 10-7
- merge order (5R1), B-8
- mode, 10-4
- mode (4R2), B-6
- name changes, B-4

# Index

---

- OM\$DEF definitions, 10-27
- other access, 10-3
- owner access, 10-3
- owner access (5R1), B-8
- paging status, 10-8
- paging status (5R1), B-9
- ring number, 10-6
- segmented, 10-9
- segmented (5R1), B-8
- sharing level (locality), 10-4
- storage, 10-7
- word boundary, 10-7, 10-9
- zero-fill, 10-9
- zero-fill (5R1), B-8
- bank type, bank attribute, 10-2
- base register environment definition (*See* BRED)
- basic mode
  - bank (BMB), B-4
  - bank field changes (4R2), B-6
  - generation of, A-2
  - link vectors, 10-22
  - OM\$DEF procedure, 10-24
  - syntax, A-4
  - with \$EQUF, A-14
  - with \$PROC directive, A-17
- BDI (*See* bank)
- BDICALL\$, 10-30
- BDIREF\$, 10-30
- b-field
  - 5-bit, A-8
  - override, A-8
- bit settings, \$TMODES (table), 8-2
- bit-by-bit operator, 4-26
- block transfer, in \$EXTEND mode, A-7
- BMB (*See* basic mode bank)
- bracket characters, in microstrings, 6-23
- BRED (base register environment definition)
  - definition of, A-8
  - with \$PROC, A-17
  - with \$USE, A-11
- BT instruction (*See* block transfer)
- built-in features
  - dictionary level, 1-2
  - directive, 1-1, 1-2
  - function, as control information, 4-23
  - node reference, 4-18
  - OS 1100, A-1

## C

### character

- conversion functions, 4-13
- conversion issues, 4-29
- in statement, 2-12
- lower case, 2-12
- manipulation functions, 4-15
- set definition, 4-10
- signed string, 5-2
- slash, as listing control, 2-14, 3-2
- stop, 2-13
- string, 2-12
- translation table, 4-11
- unsigned string, 5-3
- uppercase, 2-12
- valid, 2-12

characteristic, of floating point values, 4-7

circular data structure, D-1

### code references

- basic mode, 10-22
- extended mode, 10-23
- OM\$DEF procedure, 10-25

collating sequence, with relational operators, 4-29

### comma

- in input statement, 2-11
- in label field, 2-14
- in line item, 4-7
- in processor call, 2-2
- in selector, 4-17

### comment

- cross-reference rule, 2-8
- part of statement, 2-11, 2-12, 2-19

### common block

- 5R1 changes, B-8
- alignment attribute, 10-9
- blank, 9-6
- blank directive, 9-3
- directive, 9-2

### compatibility issues

- basic mode, A-3
- extended mode, A-3
- MASM vs ASM, C-1
- no mode, A-3
- Universal Compiling System (UCS), B-1
- with object modules, 10-29

compound relation, 4-28

### concatenated strings

- in \$DISPLAY, 3-2
- in line continuation, 2-20
- infix operator, 4-27

# Index

---

- microstring, 6-23
- with \$SR, 4-16
- condition
  - complex expression, 4-31
  - directives for testing, 6-1
  - expression, 4-30
  - operator, 4-6, 4-24
  - with \$ANDF, 6-3
  - with \$ELSE, 6-1
  - with \$ELSF, 6-2
  - with \$ENDF, 6-2
  - with \$IF, 6-1
- conditional else, 4-31
- conflicts (*See* compatibility issues)
- constant
  - floating-point, 4-7
  - integer, 4-2
  - string, 4-8
- construct
  - comma-comma, 2-18
  - space-period-space, 2-8, 2-12, 2-19
- contingency interrupt, in link vectors, 10-22
- continuation
  - line, 2-12, 2-19, 2-20
  - string, 2-20
- control
  - information, 4-23, 4-31
  - lateral transfer, 6-16
  - operator, 4-32
  - statement (transparent), 2-4
  - transfer, 6-15
- control information
  - compatibility issues, 10-29
  - definition of, 4-23
  - dictionary, 7-1
  - object module, 10-1
  - with \$FUNC, 6-11
  - with \$PROC, 6-11
- Conversational Time Sharing (CTS), 2-5
- conversion
  - character set issues, 4-29
  - indicator bits (table), 4-36
  - rules, 4-34
  - string, 4-13, 4-27
  - to ASCII string, 4-13
  - to decimal, 4-14
  - to Fielddata string, 4-14
  - to integer, 4-15
  - to string, 4-14
  - with relational operator, 4-29

conversion issues (*See* compatibility issues)  
 covered quotient operator, 4-25  
 cross-reference  
   identifiers, 2-8  
   listing, 2-7  
   output format, 2-8  
   rules, 2-7  
 CTS (*See* Conversational Time Sharing)

## D

D  
   cross-reference identifier, 2-8  
   hexadecimal, 4-8  
   postfix operator, 4-25, 4-30  
 data character set, definition of, 4-10  
 data generation, 5-1  
 data references  
   basic mode, 10-23  
   extended mode, 10-23  
   OM\$DEF procedure, 10-25  
 data structure  
   link vector, 10-22  
   recursive, D-1  
 data type  
   definition of, 4-1  
   interrogation functions, 4-34  
   numbers (table), 4-35  
 D-bank, minimum specification, 9-3  
 decimal  
   conversion, 4-14  
   floating-point value, 4-7  
   integer values, 4-2  
   point, 4-8  
 default  
   attribute values, 10-1, 10-27  
   bank, B-6  
 defined selector, with \$SN, 4-23  
 definition  
   \$INFO group 5, 9-4  
   attributes, 10-1, 10-19  
   element (OM\$DEF), 10-1, 10-27  
   external set, 12-2  
   field changes (4R2), B-8  
   field changes (5R1), B-9  
   field changes (6R1), B-10  
   implicit, 4-2  
   include directive, 12-3  
   link vector, 10-1  
   standard name changes, B-4

# Index

---

- definition attribute
  - address (5R1), B-9
  - address type, 10-20
  - default values, 10-21, 10-27
  - definition type, 10-19
  - definition type (5R1), B-9
  - OM\$DEF definitions, 10-28
  - SCS conformance, 10-20
  - SCS conformance (4R2), B-8
- definition mode assembly
  - compatability issues, D-2
  - description of, 12-1
  - omnibus output element, 2-2
  - redefine instruction mnemonics, A-2
  - with \$LEVEL directive, 7-4
- definition type
  - 5R1 changes, B-9
  - definition attribute, 10-19
- destination address (BT), A-7
- diagnostics, error and warning, 13-1
- dictionary
  - built-in directives, 1-2, 7-1
  - class index, 7-2
  - control item (\*), 2-14, 2-16
  - cross-reference, 2-7
  - current level of, 7-5
  - definition mode assembly, 12-1
  - description, 1-2
  - dynamic nesting level, 6-13
  - function level, 6-12
  - functions, 1-2, 7-1
  - insertion specifications, 2-14
  - level, 1-2, 7-1
  - level control, 7-4
  - library search, 2-21, 2-22
  - operation mnemonics, 1-2, 7-1
  - principal level of, 7-4
  - structure, 1-2
  - value retrieval, 1-2
  - with \$GO, 6-16
- difference operator, 4-25
- digit
  - in symbol, 2-12
  - valid, 4-2
- directive
  - \$ANDF, 6-3
  - \$ASCII, 4-10
  - \$BANK, 10-2, 10-9, 10-12, 10-22, 10-27
  - \$BANK example, 10-13
  - \$BASE, A-8
  - \$BASIC, A-4
  - \$CHAR, 4-10

\$CROSSREF, 3-3  
 \$DEF, 12-1, 12-2  
 \$DELETE, 7-2  
 \$DISPLAY, 3-1  
 \$DO, 6-4  
 \$EJECT, 3-3  
 \$ELSE, 6-1  
 \$ELSF, 6-2  
 \$END, 6-10, 10-30  
 \$ENDD, 6-5  
 \$ENDF, 6-2, C-4  
 \$ENDI, 6-6  
 \$ENDR, 6-5  
 \$EQU, 4-1  
 \$EQUF, 4-2, A-14  
 \$EXPORT, 2-16, 10-18, 10-19, 10-20, 10-27  
 \$EXPORT example, 10-21  
 \$EXTEND, A-5  
 \$FDATA, 4-10, C-4  
 \$FORCE, A-17  
 \$FORCEOFF, A-17  
 \$FORM, 4-2, 5-4, B-4  
 \$FUNC, 6-13  
 \$GEN, 5-3  
 \$GFORM, 5-5  
 \$GO, 6-6  
 \$HDG, 3-4  
 \$HEX, 3-3  
 \$IF, 6-1, C-4  
 \$IMPORT, 10-13, 10-16, 10-22, 10-27, 10-29  
 \$IMPORT example, 10-18  
 \$INCLUDE, 12-3  
 \$INFO, 9-1  
 \$INFO group 1, 9-1, 10-29, C-2  
 \$INFO group 10, 9-5  
 \$INFO group 11, 9-5, 10-29  
 \$INFO group 12, 9-6  
 \$INFO group 2, 9-2, C-2  
 \$INFO group 3, 9-3, C-2  
 \$INFO group 4, 9-3  
 \$INFO group 5, 9-4  
 \$INFO group 6, 9-4  
 \$INFO group 7, 9-4  
 \$INFO group 8, 9-5  
 \$INFO group 9, 9-5  
 \$INFO restrictions, 9-6  
 \$INSERT, 6-7  
 \$LEVEL, 7-4  
 \$LIST, 3-1  
 \$LIT, 5-8, A-12  
 \$MACH, A-18  
 \$NAME, 6-7

# Index

---

- \$NEG, 5-9
- \$NIL, 6-6
- \$OBJ, 10-1
- \$OCTAL, 3-3
- \$OUTPUT, 11-1
- \$PROC, 6-9, A-17
- \$REL, 9-1
- \$REPEAT, 6-5
- \$RES, 5-6
- \$SYM, 11-1
- \$UNLIST, 3-1
- \$USE, A-10
- \$WRD, 5-8
- built-in, 1-1
- condition testing, 6-1
- control information, 4-23
- cross-reference rule, 2-8
- determine output, 2-2
- FIELDATA, C-4
- generation mode, A-2
- looping, 6-4
- NEG, C-2
- OFF, C-4
- ON, C-4
- output determination, 2-2
- SETMIN, C-2
- summary (table), 1-3
- symbol lookup sequence, 2-22
- TYPE, C-2
- displacement field
  - with \$BASE, A-8
  - with BT instruction, A-7
- division remainder operator, 4-25
- dollar sign symbol, 2-12
- double precision
  - control, 4-30
  - in string conversion, 4-28
  - integer values, 4-2
- dropped label, 13-2
- dynamic flag, compatibility issues, 10-29
- dynamic nesting
  - of procedures, 6-13
  - subassemblies, 1-2

## E

- EDIT 1100 and MASM, 2-5
- EI\$ form
  - b-field override, A-8
  - description of, A-14
  - immediate operands, A-7



- with integer values, 4-2
- eject page, 2-14, 3-2
- END MASM line
  - description of, 2-6
  - diagnostic count, 13-2
- entry point
  - 5R1 changes, B-9
  - definition, 9-4
  - dropped label, 13-2
  - procedure, 6-10
  - procedure with \$FN, 6-22
  - table in preamble, 2-16
- equal sign operator, 4-28
- ER MCORE\$ requests, 2-6
- error detection
  - permissiveness, C-3
  - undetected, 4-31
- error flag
  - END MASM summary, 2-6
  - output line, 2-5
  - relocation, 4-26
  - truncation, 4-26
  - value, 4-26
- even starting address directive, 9-4
- exclusive OR operator, 4-26
- expression
  - as control information, 4-24
  - complex conditional, 4-31
  - conditional, 4-30
  - context of, 4-24
  - data generation, 5-1
  - data type, 4-34
  - indicator bits (table), 4-36
  - line item, 4-7
  - literal, 4-6
  - parenthetic, 4-5
- extended mode
  - bank (EMB), B-4
  - bank field changes (4R2), B-6
  - example, 10-23, 10-26
  - generation of, A-2
  - immediate operands, A-7
  - link vectors, 10-23
  - location counter, 9-5
  - OM\$DEF procedure, 10-24
  - syntax, A-5
  - with \$EQUF, A-15
  - with \$PROC directive, A-17
- external definitions, set of, 12-1
- external reference
  - attributes, 10-13
  - definition (group 5), 9-4

# Index

---

- during relocation, 4-3
- with \$IMPORT (4R2), B-8

externalized

- label, 2-16, 12-2
- symbol, 6-13

## F

F cross-reference identifier, 2-8

field

- \$BANK changes (4R2), B-6
- \$BANK changes (5R1), B-8
- \$BANK changes (6R1), B-9
- \$EXPORT changes (4R2), B-8
- \$EXPORT changes (5R1), B-9
- \$EXPORT changes (6R1), B-10
- \$IMPORT changes (4R2), B-7
- \$IMPORT changes (5R1), B-9
- \$IMPORT changes (6R1), B-10
- comment, 2-19
- displacement (\$BASE), A-8
- displacement (BT), A-7
- label, 2-14
- of a statement, 2-11
- operation, 2-17, 4-7
- output, 2-5

Fieldata

- at initialization, 6-23
- conversion function, 4-14
- input, 1-1
- string limits, 4-9
- with \$CHAR, 4-10, 4-11
- with \$DATE, 8-1

fieldata, underscore character, 2-13

FIELDDATA directive, C-4

fixed-point scaling operator, 4-25

flag

- attribute, 4-3
- dynamic, 10-29
- error, 2-5, 2-6, 4-26
- in operand subfield, 2-18
- leading asterisk, 4-3
- operator, 4-32
- shared, 10-29
- warning, 2-6

floating-point

- conversion, 4-34
- operator, 4-25
- scaling operator, 4-25
- value, 4-7

## form

- bank attributes, 10-11, 10-12
- definition attributes, 10-21
- line item reference, 4-7
- name changes, B-4
- OM\$DEF definitions, 10-27
- OM\$DEF in UCS, B-1
- reference attributes, 10-17
- shorter than 36 bits, C-4
- with \$BA (table), 4-4
- with \$BANK, 10-11, 10-12
- with \$EXPORT, 10-21
- with \$IMPORT, 10-17
- with literal item, 4-6
- with logical operator, 4-26

form EI\$ (*See* EI\$ form)

form I\$ (*See* I\$ form)

forward references, C-4

## function

- \$(e), 5-6
- \$AP, 4-4
- \$BA, 4-4
- \$BREG, A-12
- \$CAS, 4-13
- \$CB, 4-15
- \$CD, 4-14
- \$CFS, 4-14
- \$CS, 4-14
- \$DATE, 8-1
- \$FN, 6-22
- \$FP, 6-20
- \$FUNC directive, 6-13
- \$GP, 6-20
- \$HASH, 7-2
- \$IBITS, 4-32, 4-36
- \$IC, 7-3
- \$ILCN, 5-7
- \$L0, 4-22
- \$L1, 4-22
- \$LCB, 5-7
- \$LCFV, 5-8
- \$LCN, 5-7, C-1
- \$LCV, 5-7
- \$LEV, 7-4
- \$LF, 6-14
- \$LINES, 8-2
- \$LP, 6-20
- \$NODE, 4-22
- \$NS, 4-23
- \$PAR, 8-2
- \$SL, 4-16
- \$SN, 4-23

## Index

---

- \$SR, 4-16
- \$SS, 4-16
- \$SSS, 4-17
- \$TBIN (table), 4-35
- \$TCON (table), 4-35
- \$TDAT (table), 4-35
- \$TDIR (table), 4-35
- \$TFLT (table), 4-36
- \$TFNM (table), 4-36
- \$TFUN (table), 4-36
- \$TINM (table), 4-36
- \$TMACH, A-20
- \$TMODES, 8-2
- \$TNAM (table), 4-36
- \$TNOD (table), 4-36
- \$TPNM (table), 4-36
- \$TSTR (table), 4-36
- \$TVAL (table), 4-36
- \$TYPE, 4-35
- \$XLEV, 7-5
- built-in, 1-1
- control information, 4-23
- description of, 6-11
- dictionary, 7-1
- dictionary level, 1-2
- pass-determination (table), 6-20
- string conversion, 4-13
- string manipulation, 4-15
- summary (table), 1-5
- type testing, 4-35
- functional part of statement
  - definition of, 2-11
  - termination of, 2-20

## G

- GATED\$ONLY attribute, B-6
- general access permission, 10-3, 10-6
- generation mode
  - directives, A-2
  - with \$PROC, A-17
- generative pass
  - \$LP control function, 6-20
  - definition of, 1-1
  - with \$FP, \$GP, \$LP, 6-20
  - with \$INCLUDE, 12-3

global relational operators, C-3  
greater than equal to, operator, 4-29  
greater than, operator, 4-29

## H

hash algorithm, C-2  
hash checking level  
    5R1 changes, B-9  
    bank attribute, 10-9  
hexadecimal value  
    double-precision, 4-2  
    floating-point, 4-7  
    on output line, 2-5  
    with scaling operator, 4-26  
hierarchy of operations (*See* operator, precedence)

## I

I cross-reference identifier, 2-8  
I\$ form  
    description of, A-14  
    immediate operands, A-7  
    with integer values, 4-2  
    with shift and jump instructions, A-7  
IBJ\$/DBJ\$, use of, 10-30  
identifier  
    class of, 7-3  
    undefined, 4-1  
identity  
    node operator, 4-29  
    with node reference, 4-18  
if-then operator, 4-31  
immediate operands, in extended mode, A-7  
implicit definition, 4-2  
incompatibilities with ASM (*See* compatibility issues)  
indicator bits (table), 4-36  
indirect address, unwanted generation, C-3  
infix  
    notation, 4-24  
    string operator, 4-27  
inhibit output listing, 3-1  
initialization  
    bank attribute, 10-9  
    for assembly passes, 1-1  
    for successive calls, 2-4  
    OM\$DEF procedure, 10-24  
input (*See* MASM input)  
instruction  
    BT, A-7

# Index

---

- jump, A-7
- jump (with EI\$), A-14
- redefine, A-1
- set, for bank, 10-4
- set, predefined, 1-1
- shift, A-7
- instruction mnemonic
  - cross-reference rule, 2-8
  - for overlap registers, C-1
- integer
  - arithmetic, 4-25
  - attributes, 4-4
  - conversion, 4-15, 4-34
  - format, 4-2
  - logical operations, 4-26
  - operator, 4-24, 4-25
  - scaling operator, 4-25
  - selector, 4-17
  - value, 4-2
  - with \$CAS, 4-13
  - with \$CB, 4-15
  - with \$CD, 4-14
  - with \$CFS, 4-14
  - with \$NS, 4-23
  - with \$SN, 4-23
  - with \$SR, 4-16
  - with \$SS, 4-16
  - with \$SSS, 4-17
- Interactive Processing Facility (IPF 1100), 2-5
- internal name, definition of, 6-16
- interrupt, contingency (*See* contingency interrupt)
- IPF 1100 (*See* Interactive Processing Facility)
- ISP local priority, bank attribute, 10-7
- iteration variables, C-3

## J

- jump instruction
  - with EI\$, A-14
  - with modes, A-7

## L

- L cross-reference identifier, 2-8
- label
  - with \$(e), 5-6
  - definition, cross-reference, 2-7, 2-8
  - dictionary, 7-1
  - dropped, 13-2
  - externalized, 2-16, 12-2

- field, 2-11, 2-14
- integer item, 4-2
- literal pool, 5-8
- node reference, 4-17
- relocatable, 4-2
- selector, 2-16
- string limits, 4-9
- waiting, 6-14, 6-18
- with \$DO, 6-4
- with \$EQU, 4-1
- with \$FORM, 5-4
- with \$FUNC, 6-11
- with \$PROC, 6-8, 6-11
- words-given procedure, 6-18

lateral transfer, definition of, 6-16

leading asterisk flag, 4-3

left-justify, with \$SS, 4-16

less than equal to, operator, 4-29

less than, operator, 4-29

level

- control in dictionary, 7-4
- current, 7-5
- dictionary, 1-2, 7-1
- operators, 4-6
- principal, 7-4

levelers

- description of, 6-24
- in label field, 2-14

lexical ordering, 4-29

LIBMAX, in library search, 2-21

library

- ASM vs MASM, C-4
- definition element, 10-27
- search chain, 2-20, 2-21, 10-17
- search file, 9-6
- search order, 2-22, 9-6
- search table, 2-21, 2-23
- search time required, A-1
- searching, 2-20

limits, of strings, 4-9

line

- continuation character, 2-12, 2-19, 2-20
- input, 2-11
- item, 4-7
- levelers, 2-14
- literal, 4-6
- maximum length, 2-12
- terminator, 2-12

line number

- cross-reference identifier, 2-8
- on output line, 2-5
- segmented, 2-5

# Index

---

## link vector

- bank, 10-13, 10-22
- basic mode, 10-22
- description of, 10-22
- extended mode, 10-23
- link fault, 10-22
- OM\$DEF procedure, 10-24
- with resolution type, 10-15

Linking System, and \$IMPORT directive, 10-17

## list

- with \$L0, 4-22
- with \$L1, 4-22

## listing

- \$GP control function, 6-20
- control example, 3-5
- control of, 3-1
- cross-reference, 2-7
- diagnostic flags, 13-2
- inhibit printing, 3-1
- page control, 2-14, 3-2
- resume printing, 3-1

## literal

- creation, 4-32
- expression, 4-6
- expression (\$LIT), A-12
- pool definition, 5-8

local store priority, 4R2 changes, B-7

## locality

- 4R2 changes, B-6
- bank attribute, 10-4

## location counter

- bank field changes (4R2), B-6
- bank name, 10-12
- base function, 5-7
- base register, A-9
- blank common block, 9-3, 9-6
- blocked, 5-8, 6-11, 6-12
- common block, 9-2
- control, 6-15
- current, 2-14
- current number, C-1
- dictionary, 7-1
- during relocation, 4-3
- even, 10-12
- extended mode, 9-5
- final value function, 5-8
- initial number function, 5-7
- line item reference, 4-7
- link vector, 10-22
- literal, A-12
- local to a procedure, C-1
- number, 2-5



- number function, 5-7
- odd, 10-12
- procedure control, 6-15
- read-only, 9-5
- restoration code, C-1
- restrictions, 9-6
- specification, 2-14, 5-6
- static diagnostic information, 9-5
- synonym with bank, 10-2
- value function, 5-6, 5-7
- with \$ILCN, 5-7
- with \$IMPORT (4R2), B-8
- with \$PROC, 6-9
- words-given procedure, 6-18
- zero, 10-12
- logical
  - operation, 4-26
  - truth table (table), 4-26
- looping directive
  - \$DO, 6-4
  - \$ENDD, 6-5
  - \$ENDI, 6-6
  - \$ENDR, 6-5
  - \$REPEAT, 6-5
- lower address, bank attribute, 10-8
- lowercase character, in symbols, 2-12

## M

- M option, in search order, 2-22
- main assembly, 1-1
- manipulation of strings, 4-15
- mantissa, in floating-point values, 4-7
- MASM
  - directive summary (table), 1-3
  - function summary (table), 1-5
  - incompatibilities with ASM, C-1
- MASM input
  - ASCII, 1-1
  - character constant, 1-1
  - comment part, 2-11, 2-19
  - externalized label, 2-16
  - Fieldata, 1-1
  - functional part, 2-11
  - label field, 2-11, 2-14
  - library search, 2-22
  - line continuation, 2-19, 2-20
  - operand, 2-18
  - operand field, 2-11
  - operation field, 2-11, 2-17, 4-23
  - si, 2-1

# Index

---

- statement parts, 2-11
- updated source, 2-5
- MASM output
  - compatibility issues, 10-29
  - cross-reference format, 2-8
  - definition mode assembly, 7-4, 12-1
  - diagnostics, 13-1
  - fields, 2-5
  - library search, 2-22
  - object module element, 1-1, 2-2, 10-1
  - omnibus element, 2-2, 12-1
  - preamble, 2-5, 2-6
  - relocatable binary element, 1-1, 2-2, 9-1
  - ro, 2-2
  - so, 2-2
  - symbolic element, 2-2, 11-1
- MASM usage
  - incompatibilities with ASM, C-1
  - options summary (table), 2-2
  - processor call, 2-1
  - SIR\$ options summary (table), 2-3
  - UCS features, B-1
- MASM\$PF files, in library search, 2-21, 2-22, 2-23
- MASM\$PF1 file name, C-4
- maximum
  - address, bank attribute, 10-8
  - line length, 2-12
  - memory expansion, D-1
  - string length, 4-9
  - symbol length, 2-12
- memory restrictions, D-1
- merge order
  - 5R1 changes, B-8
  - bank attribute, 10-7
- microstring
  - description of, 6-23
  - in conditional expression, 4-30
  - substitution, 4-28
  - with \$DO, 6-5
- minimum address
  - bank attribute, 10-8
  - D-bank specification, 9-3
- minus sign
  - as arithmetic operator, 4-25
  - data generation, 5-1
  - with \$CD, 4-14
- mode
  - bank attribute, 10-4
  - bank field changes (4R2), B-6
  - directive, A-2
  - MASM operating, 8-2
  - processor settings directive, 9-1

mode basic (*See* basic mode)  
mode extended (*See* extended mode)  
mode no (*See* no mode)  
multibanked program, conversion issues, 10-30

## N

N cross-reference identifier, 2-8  
name type, with \$FN, 6-22  
NAME\$, as default, 2-2  
NEG directive, in ASM, C-2  
negation operator, 4-26  
negative  
    scaling operator, 4-25  
    transform function, 5-9  
    with \$CD, 4-14  
    with \$SR, 4-16  
nesting  
    dictionary level, 7-1  
    in two-pass procedure, 6-17  
    in words-given procedure, 6-18  
    procedure, 6-13  
    procedure walkback, 13-2  
    static, 6-24  
    subassemblies, 1-2  
    with \$DO, 6-5  
    with \$REPEAT, 6-5  
NO mode, with \$EQUF, A-14  
no mode, syntax, A-3  
node  
    identity operator, 4-29  
    \$NODE function, 4-22  
    description of, 4-17  
    flag attribute, 4-3  
    identity operator, 4-28  
    in procedure call, 6-10  
    reference, 6-11  
    selector, 2-14, 2-16  
    selector value, 4-1  
    with \$DELETE, 7-2  
    with \$FUNC, 6-11  
    with \$IC, 7-3  
    with \$L0, 4-22  
    with \$L1, 4-22  
    with \$LF, 6-14

# Index

---

nonidentity node operator, 4-30  
not equal operator, 4-29  
NOT operator, 4-26  
notation, infix/postfix/prefix, 4-24  
numeric part, with \$AP function, 4-4

## O

object module  
    bank group, 10-2  
    changes (4R2), B-6  
    changes (5R1), B-8  
    changes (6R1), B-9  
    compatibility, 10-29  
    element, 1-1, 2-5  
    entry point, 10-30  
    evolution, B-1  
    output routine (OMOR), 1-1, 10-29  
    start address, 10-30  
    with \$OBJ, 10-1  
octal value  
    floating-point, 4-7  
    on output line, 2-5  
    specification of, 4-2  
    with scaling operator, 4-26  
OFF directive  
    in MASM, C-4  
    synonym for \$ENDF, 6-2  
OM\$DEF  
    5R1 changes in SYS\$LIB\$\*MASM, B-9  
    contents, 10-27  
    name changes, B-1  
OMDEF\$ (*See* OM\$DEF)  
omnibus element, 2-2, 2-5, 12-1  
OMOR (object module output routine), 1-1, 10-29  
ON directive  
    in MASM, C-4  
    synonym for \$IF, 6-1  
one's complement negative, vs NOT operator, 4-26  
one-pass procedure  
    characteristics of, 6-16  
    with \$FP, \$GP, \$LP, 6-20  
operand  
    conversion, 4-34  
    field, 2-11  
    with node reference, 4-18  
operating mode, 8-2  
operation field  
    as control information, 4-23  
    description of, 2-11, 2-17  
    in line item, 4-7

- operation mnemonics
  - dictionary level, 1-2
  - in dictionary, 7-1
- operation, logical hierarchy (*See* operator, precedence)
- operator
  - arithmetic, 4-25
  - asterisk flag, 4-32
  - conditional, 4-6, 4-24
  - context of, 4-24
  - control information, 4-32
  - fixed-point scaling, 4-25
  - flag, 4-32
  - floating-point scaling, 4-25
  - in line item, 4-7
  - infix, 4-24
  - integer, 4-24
  - logical, 4-26
  - postfix, 4-24, 4-28
  - precedence, 4-24, 4-27, 4-33
  - precedence (table), 4-33
  - prefix, 4-24
  - relational, 4-28
  - relational, global, C-3
  - scaling, 4-25
  - string, 4-27
  - unary, 4-24
  - unary prefix, 4-32
  - with parentheses, 4-5
- options
  - listing control, 3-1
  - MASM summary (table), 2-2
  - SIR\$ summary (table), 2-3
- OR operator, 4-26
- ordinal integer, 4-23
- OS 1100, features, A-1
- other access permission, bank attribute, 10-3
- output
  - (*See also* MASM output)
  - object module elements, 10-1
  - types, 2-4, 9-1
- overflow arithmetic, C-2
- overlap registers, C-1
- owner access, 5R1 changes, B-8
- owner access permission, bank attribute, 10-3

## P

- P cross-reference identifier, 2-8
- page control, 2-14, 3-2
- paging status
  - 5R1 changes, B-9

# Index

---

- bank attribute, 10-8
- parameter
  - conversion rules, 4-1
  - function, 6-12
  - node reference, 4-18
  - on processor call, 8-2
  - pair, 4-11
  - procedure, 6-9
  - with \$BANK, 10-9
  - with \$DISPLAY, 3-1
  - with \$EXPORT, 10-21
  - with \$IMPORT, 10-17
  - with \$L0, 4-22
  - with \$PROC, 6-8
- parentheses
  - for precision, 4-30
  - in conditional expression, 4-31
  - in data generation, 5-1
  - in expression, 4-5
  - in line item, 4-7
  - in literal expression, 4-6
  - node selector, 2-16, 4-17
- parenthetical expression items, 4-5
- pass-determination functions (table), 6-20
- passes of assembly
  - cross-reference rule, 2-7
  - definition mode, 12-2
  - functions (table), 6-20
  - generative, 1-1
  - initialization, 6-23
  - speeding up, 6-20
  - summary, 1-1
  - two-pass procedure, 6-17
  - with \$FP, \$GP, \$LP, 6-20
- PDP (procedure definition processor), cross-reference rule, 2-7
- period
  - as decimal point, 4-7
  - in MASM statement, 2-11
- plus sign
  - as arithmetic operator, 4-25
  - data generation, 5-1
- positive scaling operator, 4-25
- postfix
  - notation, 4-24
  - operator, 4-28
  - operator, double-precision, 4-2
  - operator, precision, 4-30
- preamble
  - output, 2-5, 2-6
  - relocatable binary, 2-16

precedence of operators (*See* operator, precedence)

precision

of floating-point value, 4-7

arithmetic, 4-2, 4-25, C-2

control of, 4-30

double, 4-25

in string conversion, 4-28

single, 4-25

prefix

notation, 4-24

operator, 4-32

procedure

\$END directive, 6-10

\$PROC directive, 6-9

calling, 6-10

control information, 4-23

definition, 6-9

entry point, 6-15, 6-16, 6-22

local location counter, C-1

location counter control, 6-15

name changes, B-6

nest, 13-2

nesting, 6-13

OM\$DEF in UCS, B-1

OM\$DEF standard, 10-24

pass-determination (table), 6-20

redefinition, A-2

two-pass, 6-17, 6-20

words-given, 6-9, 6-14, 6-15, 6-18

procedure library search (*See* library, search)

procedure library search table (PLIBT), 2-23

processor

mode settings, 9-1

reusability, 2-4

processor call

input, 2-1

MASM options summary (table), 2-2

object module, 10-1

options, 2-1

output, 2-2

output (source), 2-2

SIR\$ options summary (table), 2-3

statement, 2-1

product, arithmetic operator, 4-25

## Q

quarter-word sensitive, 9-2, 10-29

quotation mark

in character string, 5-3

in microstring, 6-23

# Index

---

single, 2-20  
quoted string, cross-reference rule, 2-8  
quotient, arithmetic operator, 4-25

## R

READ\$GATED attribute, B-6  
read-only location counter directive, 9-5  
recursive data structure, D-1  
redefine instruction mnemonics, A-1  
reference  
    attributes, 10-1  
    code, basic mode, 10-22  
    code, extended mode, 10-23  
    code, OM\$DEF procedure, 10-25  
    data, basic mode, 10-23  
    data, extended mode, 10-23  
    external, 9-4, 10-13, 12-2  
    forward, C-4  
    imported, 10-13  
    link vector, 10-22  
    node, 4-17  
reference attribute  
    default values, 10-17, 10-27  
    name changes, B-4  
    OM\$DEF definitions, 10-28  
    reference type, 10-14  
    reference type (4R2), B-7  
    resolution type, 10-14  
    resolution type (4R2), B-7  
    resolution type (5R1), B-9  
    SCS conformance, 10-16  
    SCS conformance (4R2), B-7  
    storage, 10-16  
    storage (4R2), B-7  
    strength, 10-14  
    strength (4R2), B-7  
reference type  
    4R2 changes, B-7  
    reference attribute, 10-14  
register usage  
    B (extended mode), 10-23  
    in cross-reference, 2-7, 2-8  
    overlap registers, C-1  
    with \$BASE, A-8  
    with \$LIT, A-12  
    with \$PROC, A-17  
    with \$USE, A-11  
    with BT, A-7  
    with five-bit b-field, A-8  
    X (basic mode), 10-22



- relational operator
  - description of, 4-28
  - global, C-3
- relative address, definition of, 4-3
- relocatable
  - label, 4-2
  - literal items, 4-6
  - output routine (ROR), 1-1
  - with base register, A-9
- relocatable binary
  - data generation, 5-1
  - element, 1-1, 9-1
  - negative value, 5-9
  - output, 9-1
  - preamble by object module, 2-5
  - replacement for, 10-1
- relocation
  - \$BA selectors (table), 4-4
  - by external reference, 4-3, 4-4
  - by location counter, 4-3, 4-4
  - description of, 4-3
  - error, 4-26
  - information, 4-4
  - negative, 4-4
  - with \$IMPORT, 10-17
  - with \$IMPORT (4R2), B-8
  - with conversion, 4-34
  - with logical operator, 4-26
  - with relational operator, 4-29
- resolution type
  - 4R2 changes, B-7
  - 5R1 changes, B-9
  - 6R1 changes, B-10
  - reference attribute, 10-14
- resume listing, 3-1
- ring number
  - 5R1 changes, B-8
  - bank attribute, 10-6
- ro
  - in library search, 2-21
  - in processor call, 2-2
- ROR (relocatable output routine)
  - output, 1-1
  - word size, 5-9
- rules for
  - \$SSS result type, 4-17
  - operand conversion, 4-34
  - operator hierarchy, 4-24
  - parameter conversion, 4-1, 4-10, 4-22, 4-35, 4-36, 6-11, 6-22

runstream, as source default, 2-1

## S

### S

cross-reference identifier, 2-8

postfix operator, 4-25, 4-30

### sample

definition, 6-5

definition (function), 6-11

definition (procedure), 6-8

ending, 6-10

input, 6-9

nesting, 6-13

saving, 6-24

with \$ENDR, 6-5

with \$GO, 6-6

with \$NAME, 6-7

with \$PROC, 6-8, 6-9

with \$REPEAT, 6-5

with levelers, 6-24, 6-25, 6-27

scaling operators, 4-25

### SCS conformance

4R2 changes, B-7, B-8

definition attribute, 10-20

reference attribute, 10-16

SDD (*See* symbolic debugging dictionary)

search order (*See* library, search order)

### segmented

5R1 changes, B-8

bank attribute, 10-9

### selector

as label, 2-16

\$NS function, 4-23

\$SN function, 4-23

as control information, 4-24

description of, 4-17

flag attribute, 4-3

nested procedures, 6-13

node, 2-14

value, 4-1

with \$DELETE, 7-2

with \$FN, 6-22

with \$FUNC, 6-11

with \$IC, 7-3

with \$L0, 4-22

with \$L1, 4-22

with \$LF, 6-14

with \$PROC, 6-9, 6-10

with \$REPEAT, 6-5

- semicolon, as continuation character, 2-12, 2-20
- SETMIN directive, in ASM, C-2
- shared flag, 10-29
- sharing level (locality), bank attribute, 10-4
- shift
  - arithmetic, 4-25
  - instructions, A-7
- si
  - in library search, 2-21
  - in processor call, 2-1
- signed character strings, 5-2
- significant digit
  - with \$CB, 4-15
  - with \$CD, 4-14
- single precision
  - control, 4-30
  - in string conversion, 4-28
  - integer values, 4-2
  - with logical operator, 4-26
- SIR\$ (symbolic input/output routine)
  - MASM usage, 1-1
  - options summary (table), 2-3
- SIR\$ (symbolic input/output routine)
  - options on processor call, 2-1
  - with \$SYM directive, 11-1
- slash
  - as arithmetic operator, 4-25
  - control information, 4-24
  - in conditional expression, 4-31, 4-32
  - page eject, 2-14, 3-2
  - unary operator, 4-32
- so
  - in library search, 2-21
  - in processor call, 2-2
- sorting technique, 4-23
- source
  - address (BT), A-7
  - image on output line, 2-5
  - input (si), 2-1
  - input (updated), 2-5
  - line numbers, 2-5
  - output (so), 2-2
- space
  - cross-reference identifier, 2-8
  - in input statement, 2-11
  - in line item, 4-7
  - in processor call, 2-2
  - with \$CB, 4-15
  - with \$SSS, 4-17

# Index

---

- space-period-space construct, 2-8, 2-12, 2-19
- special access permission, 10-3, 10-6
- standard definition, name changes, B-4
- START\$ reserved word, 10-30
- statement
  - complex structure, 2-20
  - parts of, 2-11
- static
  - diagnostic information, 9-5
  - nesting, 6-24
- stop character, for demand terminal, 2-13
- storage
  - 4R2 changes, B-7
  - bank attribute, 10-7
  - bank mode attribute, 10-4
  - compactions, 2-6
  - location counters, 5-6
  - pool, 2-6
  - reference attribute, 10-16
  - relative address, 4-3
  - reserve space directive, 5-6
  - usage summary, 2-6
  - use and reuse, 2-4
- strength
  - 4R2 changes, B-7
  - reference attribute, 10-14
- string
  - character, 2-12
  - concatenation, 2-20, 3-2, 4-27
  - constant, 4-8
  - continuation, 2-20
  - conversion, 4-27, 4-34
  - conversion functions, 4-13
  - length function, 4-16
  - limits, 4-9
  - manipulation functions, 4-15
  - microstring, 6-23
  - operator, 4-27
  - repetition function, 4-16
  - signed character, 5-2
  - symbol, 2-12
  - unsigned character, 5-3
- subassembly
  - \$END directive, 6-10
  - \$FP control function, 6-20
  - \$FUNC directive, 6-13
  - \$PROC directive, 6-9
  - called by function, 6-12
  - called by procedure, 6-8
  - definition of, 1-1
  - dictionary level, 7-1
  - dynamic nesting, 1-2

- interaction, A-17
  - with \$ILCN, 5-7
- subfield
  - in input statement, 2-11
  - value, 4-1
- subscript, usage with nodes, 4-17
- substring
  - extraction function, 4-16
  - substitution function, 4-17
- sum, arithmetic operator, 4-25
- summary pass
  - definition of, 1-1
  - with \$FP, \$GP, \$LP, 6-20
- SUP usage, in END MASM line, 2-6
- symbol
  - ampersand, C-1
  - attributes, 10-1
  - collector-defined, 10-29
  - definition, 2-12
  - dollar sign, 2-12
  - external, 4-2, 6-13, 9-4
  - in cross-reference, 2-7, 2-8
  - in dictionary, 7-2
  - in expression, 4-24
  - in label field, 2-14
  - in library search, 2-21
  - in preamble of RB, 2-5
  - lookup sequence, 2-22
  - underscore, 2-12
  - user-defined, 2-12
  - value, 4-1
- symbolic
  - debugging dictionary (SDD), 10-10, 10-12
  - name changes, B-2
  - output mode, 11-1
  - reference attributes, 10-1
- symbolic input/output routine (*See* SIR\$)
- SYS\$\*RLIB\$
  - in library search, 2-21
  - in search order, 2-22
- SYS\$LIB\$\*MASM
  - 5R1 changes, B-9
  - calling MASM, 2-1
  - in library search, 2-21
  - with object modules, 10-1
- SYS\$LIB\$\*PROC, in library search, 2-21
- SYS\$LIB\$\*SYSLIB
  - in library search, 2-21
  - in search order, 2-22
- system character set, 4-10
- system file
  - default, 2-22

# Index

---

in library search, 2-22, 2-23

## T

terminator line (*See* line terminator)

third-word sensitive, 9-2, 10-29

TPF\$, as default, 2-2

trailing D, for precision, 4-30

transfer

control, 6-15

lateral, 6-16

transformation

one's complement, 4-25

two's complement, 5-9

translation table, with \$CHAR, 4-10

tree structure

construction function, 4-22

procedure parameters, 6-9, 6-12, 6-16

truncation errors, with arithmetic operators, 4-26

truth table (table), 4-26

two-pass assembler, 1-1

two-pass procedure

characteristics of, 6-16

description of, 6-17

speeding up, 6-20

with \$FP, \$GP, \$LP, 6-20

type

output, 2-4, 9-1

selector, 4-17

testing, 4-35

testing functions (table), 4-35

TYPE directive, in ASM, C-2

## U

U cross-reference identifier, 2-8

UCS (*See* Universal Compiling System)

unary

asterisk, 4-6

asterisk (in literal item), 4-6

asterisk (set flag), 4-3

operator, 4-24

positive operator, 4-25

prefix operator, 4-32

underscore

fielddata, 2-13

in symbol, 2-12

Universal Compiling System (UCS), B-1  
unsigned character strings, 5-3  
uppercase character, in symbols, 2-12  
user-defined symbol (*See* symbol, user-defined)

## V

value  
    absolute part, 4-2  
    error, 4-26  
    flagged, 4-3  
    floating-point, 4-7  
    integer, 4-2  
    relocation, 4-2  
    retrieve from dictionary, 1-2  
    selector, 4-17  
    string, 4-8  
    types of, 4-1  
variable, iteration (*See* iteration variable)  
void  
    bank, 9-5, 10-2  
    field, in statement, 2-11  
    in conditional expression, 4-31  
    microstring, 6-24  
    operation field, 2-17, 4-7  
    with \$CHAR, 4-10  
    with \$SR, 4-16  
    with \$SS, 4-16  
    with \$SSS, 4-17

## W

waiting label  
    description of, 6-14  
    externalized, 6-13  
    with \$LF, 6-14  
    words-given procedure, 6-18  
warning flag  
    summary (table), 13-1  
    summary listing, 2-6  
word  
    boundary attribute, 10-7, 10-9  
    single-precision, 4-2  
    size specification directive, 5-8  
    string length, 4-9  
word generation  
    signed strings, 5-2  
    unsigned strings, 5-3  
words-given procedure  
    characteristics of, 6-16

## Index

---

- description of, 6-18
- with \$FP, \$GP, \$LP, 6-20
- with the \$PROC directive, 6-9
- with waiting labels, 6-14

## X

- X
  - cross-reference identifier, 2-8
  - register, 10-22

## Z

- zero
  - division remainder, 4-25
  - in conditional expression, 4-31
  - in floating-point number, 4-8
  - location counter, 5-6
  - with \$CB, 4-15
  - with \$SN, 4-23
  - with \$SR, 4-16
  - with \$SS, 4-16
  - with \$SSS, 4-17
  - with NOT operator, 4-26
  - with relational operator, 4-28
  - with type testing function, 4-35
- zero-fill
  - 5R1 changes, B-8
  - bank attribute, 10-2, 10-9